Validation formelle des langages à parallélisme de données

David Cachera

N° d'ordre : 81

THÈSE

présentée devant

l'École normale supérieure de Lyon

pour obtenir le titre de

Docteur de l'École normale supérieure de Lyon
Spécialité : Informatique

au titre de la formation doctorale : Informatique de Lyon

par David CACHERA

Titre :

Validation formelle des langages à parallélisme de données

Soutenue le 8 janvier 1998

Après avis de : Monsieur Dominique MÉRY
Monsieur Patrice QUINTON

Devant la commission d'examen formée de :

Monsieur Luc BOUGÉ (directeur de thèse)
Madame Susanne GRAF
Monsieur Dominique MÉRY
Monsieur Antoine PETIT (président du jury)
Monsieur Patrice QUINTON

Que pourra-t-on bien faire en effet, au jour du Jugement dernier, quand seront pesées les œuvres humaines, de trois traités sur l'acide formique, ou même de trente, s'il le fallait? D'autre part, que peut-on savoir du Jugement dernier si l'on ne sait même pas tout ce qui peut d'ici là sortir de l'acide formique?

Robert Musil, *L'Homme sans qualités.*

Remerciements

Je tiens tout d'abord à remercier Antoine Petit de m'avoir fait l'honneur de présider ce jury.

Merci à Dominique Méry et Patrice Quinton pour l'intérêt qu'ils ont porté à ce travail et le temps qu'ils y ont consacré.

Merci à Susanne Graf pour sa lecture très attentive du manuscrit et ses nombreuses remarques témoignant d'un vif intérêt.

C'est un nombre potentiellement infini de remerciements qu'il me faut adresser à Luc Bougé, qui a encadré ce travail. Pour avoir insufflé la vie à ces recherches, pour m'avoir toujours gardé sur le chemin de la rigueur, et pour son infinie patience.

Merci aux membres de l'équipe API de l'IRISA avec lesquels j'ai eu des contacts plus que fructueux, en particulier Sanjay et Florent Dupont Papa de D.

Les membres du département d'informatique de Chalmers m'ont accueilli et soutenu pendant la « dernière ligne droite ». Qu'ils en soient chaleureusement remerciés, et tout spécialement Mary.

Merci aux secrétaires, du LIP et d'ailleurs, pour leur efficacité et leur bonne humeur. Je remercie aussi tous les membres du laboratoire, sans qui la maison ne serait pas aussi chaude et accueillante, et plus particulièrement tous ceux à qui j'ai dû un jour céder une parcelle de mon bureau : Jean-François, Gil, Christian, Richard, Matthieu, et enfin Frédo, qui, si SON passage dans MON bureau fut des plus éphémères — au moins officiellement —, a été de loin le plus sollicité.

Merci également à ceux qui ont dû me supporter au quotidien durant ces années : Sophie, Mathias, Tristan, Åse et Christophe. Ça n'a pas toujours été facile pour eux...

Merci finalement à mes parents, à Laurence, aux membres de ma famille, et à tous ceux qui m'ont soutenu ou aidé, ponctuellement ou durablement, de loin ou de près, ceux qui ont arrosé les plantes, ceux qui ont gardé le chat, ceux qui ont coupé le saucisson, ceux qui étaient en régie pour lancer la bande son, ceux qui ont réglé les éclairages, et ceux qui n'ont rien fait mais n'en pensaient pas moins.

Table des matières

Introduction

De nombreux efforts ont été mis en œuvre dans les trois dernières décennies pour développer des machines parallèles de plus en plus puissantes. De nombreux modèles d'*exécution* ont été proposés pour ces machines. La terminologie de Flynn ([32]) propose de classer ceux-ci, qu'ils concernent le calcul séquentiel ou parallèle, en quatre catégories, selon que les flots de Données et d'Instructions (contrôle) sont Simples ou Multiples. En ce qui concerne les machines parallèles, cette classification distingue deux types d'architectures : SIMD (machines massivement parallèles de type MasPar) et MIMD (machines multiprocesseurs, réseaux de stations, etc.).

L'histoire du développement des machines et langages séquentiels montre que, à partir du modèle d'exécution quasi universel de Von Neumann, de nombreux modèles de *programmation* ont été proposés. Ceux-ci ont évolué des langages « bas niveau » (du type Fortran) vers des langages davantage structurés et abstraits (de Pascal aux langages de programmation logique, fonctionnels ou à objets). L'évolution du parallélisme atteint ce point depuis peu : les premières machines étaient quasiment toutes « livrées » avec un modèle de programmation différent, qui était en fait un langage très proche de l'architecture. Le besoin de concevoir des applications portables nécessite pourtant la présence de modèles de programmation plus universels. Deux modèles de programmation, directement inspirés des modèles d'exécution MIMD et SIMD, sont apparus. D'un côté, l'abstraction du modèle MIMD a donné naissance au modèle de programmation du parallélisme de contrôle, illustré par les CSP de Hoare [41], puis des langages comme Occam, ou des bibliothèques de communication comme PVM ou MPI. De l'autre côté, le modèle SIMD a donné naissance au modèle de programmation du parallélisme de données.

Ces modèles se sont peu à peu détachés du modèle d'exécution sous-jacent. Un langage à parallélisme de données permettra par exemple de programmer à la fois un réseau de stations ou une matrice de processeurs. La distinction s'opère plutôt maintenant au niveau du type d'applications et de la granularité du parallélisme : un langage à parallélisme de données sera plutôt destiné à programmer des applications régulières, à grain fin de parallélisme (typiquement du calcul scientifique), alors que le parallélisme de contrôle sera bien adapté à des applications à gros grain de parallélisme. En ce qui concerne le parallélisme massif, le modèle du parallélisme de contrôle paraît difficile à mettre en œuvre, tant au niveau de la conception que du débogage ou de la validation. Il devient en effet rapidement très compliqué de gérer des milliers de processus. En revanche, le modèle du parallélisme de données, de par sa relative simplicité, semble actuellement prometteur. Cette thèse se penche sur le problème de la validation de langages pour le parallélisme massif. Nous nous sommes donc naturellement tourné vers ce dernier modèle.

1 Langages à parallélisme de données

Le modèle de programmation à parallélisme de données a donné naissance à une classe de langages de même nom, encore appelés langages *data-parallèles*. Certains langages sont uniquement

à parallélisme de données *explicite*. Dans ces langages, le problème de l'expression du parallélisme doit être géré par le programmeur/utilisateur. C'est le cas de plusieurs langages dérivés de C, comme C* [67] ou HyperC [57]. Un programme est une séquence d'instructions parallèles, qui sont soit des opérations point-à-point entre objets parallèles, soit des opérations de réarrangement. Par exemple, en C*, si A, B et C sont des variables définies sur le domaine $[1..N]$, l'instruction suivante

```
A = B + C
```

affecte à toutes les composantes de A les sommes respectives des composantes de B et de C. Avant d'effectuer cette instruction, l'utilisateur a dû spécifier explicitement les domaines spatiaux de ses variables. L'instruction suivante

```
A = [.+1]A
```

quant à elle permet d'effectuer un décalage vers la gauche des composantes de A. Cette instruction est qualifiée d'explicite dans le sens où l'utilisateur spécifie lui-même la fonction de communication.

Par ailleurs, si l'on comprend la notion de parallélisme de données au sens large, la classe des langages qu'elle représente est très vaste, puisqu'elle peut aller jusqu'à contenir tous les langages où le parallélisme est implicite. Considérons par exemple le programme Fortran suivant

```
   DO 10 I=1,N
      A[I]=B[I]+C[I]
10 CONTINUE
```

Ce programme peut être exécuté en parallèle avec une seule instruction s'exécutant à la fois sur tous les éléments des tableaux concernés, même si a priori les variables sont définies sur des tableaux sans notion de domaine spatial. Son « comportement » sera alors identique à celui du programme C* précédent. Extraire le parallélisme d'un tel programme est le but de la parallélisation automatique : les compilateurs-paralléliseurs traduisent un programme séquentiel en un exécutable pour machines parallèles. Il s'agit de trouver le placement des calculs dans le temps (ordonnancement) et l'espace (allocation). De vastes travaux ont été menés dans ce domaine de la parallélisation, que ce soit pour les langages impératifs (voir par exemple [24] ou [28]), ou pour d'autres types de langages comme par exemple les langages à flot de données synchrones (voir par exemple [34]). Ce problème est résolu pour certaines classes réduites de programmes (par exemple les nids de boucles parfaits des programmes de type Fortran), mais reste difficile dans des cas plus généraux.

De nombreuses versions de Fortran ont été développées (CM Fortran, Fortran D, Vienna Fortran, etc.), proposant plus ou moins de parallélisme. Un pas vers une unification a été franchi avec l'élaboration de la norme High Performance Fortran (HPF, [44]). Avec ce dernier langage, l'utilisateur peut lui-même spécifier des instructions de contrôle *explicitement* parallèles, avec par exemple la structure de contrôle `FORALL`, et également le placement des données, avec les directives de compilation `ALIGN` et `DISTRIBUTE`. Les langages à parallélisme de données constituent donc un vaste ensemble, des plus « explicites » aux plus « implicites », avec de nombreuses gradations entre ces deux extrêmes.

2 Deux approches pour la validation formelle

Le modèle de programmation à parallélisme de contrôle a bénéficié d'un effort important vers le développement de méthodes de validation formelle. La complexité des applications envisagées

rend en effet indispensable une telle validation. Malgré leur relative simplicité, les langages data-parallèles ont été peu étudiés de ce point de vue. Dans cette thèse, nous essaierons de montrer que la validation de langages data-parallèles est du même niveau de « complexité » que celle des langages séquentiels scalaires. Si nous laissons de côté les méthodes de *model-checking* qui s'appliquent aux systèmes d'états finis, nous pouvons distinguer entre deux grandes classes de méthodes de validation formelle.

- L'approche transformationnelle, que nous appelerons également approche *interne*. Cette approche procède par réécritures successives des programmes, selon des règles garantissant une certaine équivalence. Programmes et spécifications sont donc écrits dans le même langage. La correction du programme final est garantie par la correction des règles de réécriture et par celle de la spécification initiale.

- L'approche assertionnelle, que nous appelerons également approche *externe*. Cette approche utilise un *langage d'assertion* différent du langage de programmation. Une méthode de validation consiste à établir un lien entre programmes et spécifications.

Dans le cas des langages impératifs scalaires, la balance penche clairement en faveur de la seconde approche, avec notamment la logique de Hoare [40]. Les langages fonctionnels s'avèrent en revanche particulièrement favorables à la première. Des modèles comme Unity [18] tirent quant à eux parti des deux méthodes.

Dans le cadre de cette thèse, nous allons exclusivement nous intéresser à l'approche externe, en l'appliquant à deux langages. Le premier est un langage impératif, le second un langage fonctionnel.

3 Parallélisme explicite et langage impératif

Le premier langage auquel nous nous sommes intéressé est un langage impératif, où le parallélisme de données est explicite. Ce langage, nommé $\mathcal{L}$, est un langage « jouet », conçu comme un noyau commun à plusieurs langages existants comme C* ou MPL. Les méthodes externes de validation formelle de langages de ce type ont déjà fait l'objet de quelques travaux. Nous les passons brièvement en revue[1].

Le langage VAL de Clint et Narayana

Une des premières tentatives de construction d'un système de preuve pour un langage data-parallèle fut celle de Narayana et Clint en 1983 [20, 21]. Ceux-ci utilisent le langage VAL, inspiré d'ACTUS. ACTUS [58] a été défini par Perrott, dans le but de créer un langage parallèle de haut niveau utilisable à la fois pour des tableaux de processeurs ou pour des processeurs vectoriels. Sa syntaxe est semblable à celle de PASCAL. Les grands principes qui ont dirigé la définition de ce langage sont les suivants.

- La présence du parallélisme dans la syntaxe : l'utilisateur exprime lui-même le parallélisme de son problème .

- La possibilité de faire varier le contexte, c'est-à-dire l'ensemble des indices pour lesquels une variable est active.

1. Outre les références données dans le texte, une description assez complète de ces approches peut être trouvée dans la thèse de Utard : Sémantique des langages à parallélisme de données. Applications à la validation et à la compilation, 1995 ([68]).

- Le contrôle du parallélisme non seulement par les données, suivant le modèle data-parallèle, mais aussi de façon explicite.

Clint et Narayana ont repris dans VAL les principales caractéristiques d'ACTUS. Les objets de base de ce langage sont d'une part les tableaux, c'est-à-dire les variables parallèles, et les séquences d'accès, qui sont des ensembles ordonnés monotones et finis d'entiers. La séquence d'accès associée à une variable permet de déterminer quels indices de cette variable vont être utilisés, et correspond donc au contexte d'activité de celle-ci. Il est possible de définir des séquences d'accès arbitraires, ou d'utiliser des opérateurs de sélection ou de décalage sur les séquences. Le contexte peut parfois être déterminé de façon dynamique, par exemple dans une boucle ou une structure de conditionnement. Il existe donc un symbole spécial, #, qui désigne le contexte courant (déterminé par la structure englobante) associé à la variable considérée. Par exemple, si x est une variable définie sur le domaine `[1:5]`, et y une variable définie sur le domaine `[6:10]`, l'expression booléenne `x[1:5] >y[6:10]` compare deux-à-deux les valeurs x_i et y_{i+5}. Le programme suivant

```
if ( x[1:5] > y[6:10] ) then for x,y do x[#] := y[#];
```

mettra à jour, pour tous les indices i pour lesquels l'expression booléenne est vraie, la valeur de x_i avec la valeur de y_{i+5} (et non y_i) : le signe # représente une valeur *dynamique* du contexte, mais qui reste *locale* à chaque variable.

Clint et Narayana ont développé pour ce langage un système de preuve dont il ont prouvé la complétude [20]. Dans ce système, le contexte est directement manipulé par les assertions : la séquence d'accès associée à la variable x est notée $x.as$. La possibilité de manipuler de façons diverses les séquences d'accès dans le langage, ainsi que le fait d'associer à chaque variable un contexte spécifique, donnent des règles de preuve assez complexes que nous ne détaillerons pas ici.

L'approche de Stewart

Plus récemment, Stewart [65, 71, 66] a introduit en 1988 un langage qu'il a voulu indépendant de toute considération d'implémentation. Son but a été d'isoler un ensemble de primitives nécessaires pour la construction de programmes data-parallèles, et, surtout, de donner une sémantique dénotationnelle claire et précise pour ces primitives plutôt que de favoriser une implémentation pratique ou efficace.

Le langage utilisé par Stewart sépare les structures de contrôle parallèle de la manipulation de contexte. Les constructions de ce langage sont les « instructions SIMD généralisées ». Celles-ci sont composées de deux éléments :

- une instruction I,
- un sous-ensemble S de l'ensemble des indices des processeurs.

L'instruction SIMD généralisée correspond simplement à l'application de l'instruction I sur le sous-ensemble S. Ce sous-ensemble peut être aussi complexe que l'on veut : on le construit à partir de segments d'entiers, par union, intersection, ou restriction par une condition booléenne. L'affectation peut se faire avec ou sans communication, ce dernier cas n'étant qu'un cas particulier, nommé « affectation SIMD localisée ». La communication s'effectue par l'intermédiaire de *fonctions de connexion*, qui sont des fonctions de l'ensemble des indices dans lui-même, construites par l'utilisateur et exprimées à l'aide de λ-termes. En plus de ces fonctions de communication explicites, les fonctions ANYIN et ALLIN permettent d'exprimer des OU et ET globaux sur un ensemble d'indices.

Le système de preuve utilisé par Stewart repose sur un mécanisme de *substitution généralisée* (*qualified substitution*) : intuitivement, $\langle E(i), a, S\rangle$ où E est une expression dépendant de l'indice i et contenant éventuellement une fonction de communication, a une variable parallèle et S un ensemble d'indices, représente l'ensemble des substitutions *simultanées* de $a(j)$ par $E(j)$ pour tout $j \in S$. Ainsi, l'axiome de l'affectation s'écrira

$$\{p\langle E(i), a, S\rangle\}\ \forall i \in S.\, a(i) := E(i)\ \{p\}.$$

La difficulté ici réside dans la possibilité de traiter des expressions du type

$$a(a(i))$$

où a est un tableau, ce qui donne des règles de substitution assez compliquées. Le langage proposé par Stewart possède l'avantage sur VAL de présenter une sémantique axiomatique beaucoup plus simple. Néanmoins, la construction du langage rend difficile la séparation de la communication et de l'affectation « locale », et nécessite par ailleurs de spécifier le contexte à chaque instruction.

L'approche de Gabarró et Galvaldá

Gabarró et Galvaldá [33] ont également proposé en 1994 un langage data-parallèle et une sémantique axiomatique associée. La principale construction parallèle de ce langage est la construction

$$\textsf{for all } k \textsf{ do in parallel } S(k) \textsf{ end}$$

qui signifie que les $S(k)$ doivent être exécutés en parallèle pour tout k. Cette construction se généralise en $\textsf{for all } k : E(k) \textsf{ do in parallel } S(k) \textsf{ end}$, qui signifie que seuls les $S(k)$ tels que l'expression booléenne $E(k)$ est vraie seront exécutés. Le langage possède également une structure conditionnelle $\textsf{if } E(k) \textsf{ then } S(k) \textsf{ end}$, les structures telles que séquencement, boucles, procédures, etc. héritées des langages scalaires, et une instruction d'affectation $x[F_1(k)] := F_2(k)$. Si $F_1(k) \neq k$, il y a alors communication implicite. Plus spécifiquement, on trouve aussi une structure « inconditionnelle », $\textsf{unconditionally } S \textsf{ end}$, qui permet l'exécution de S sur tous les processeurs, quel que soit le contexte courant, ainsi que les fonctions count et enumerate qui renvoient respectivement le nombre des processeurs actifs et une énumération de ceux-ci.

L'originalité de Gabarró et Galvaldá par rapport aux approches précédentes réside dans leur système de preuve. En effet, leur langage d'assertion dissocie les habituels prédicats portant sur les variables du programme de l'expression de l'ensemble des processeurs actifs. Ceci permet de traiter de façon totalement séparée les manipulations de contexte. Ainsi, la spécification d'un programme S sera de la forme

$$\{A; P\}\ S\ \{B; Q\},$$

ce qui signifie que si S débute dans le contexte où les processeurs de l'ensemble A sont actifs, et dans un état qui valide le prédicat P, alors, si S termine, il termine dans un état qui valide le prédicat Q, avec l'ensemble de processeurs B actifs (en général $A = B$).

4 Parallélisme implicite et langage équationnel

La seconde partie de cette thèse traitera du langage ALPHA. Ce langage est un langage équationnel conçu initialement pour décrire des circuits systoliques à partir du formalisme des équations récurrentes affines. Dans ce langage, l'expression du parallélisme est plutôt implicite : les objets

manipulés sont des polyèdres multidimensionnels, mais il n'y a pas de notion de contrôle qui expliciterait le parallélisme.

Le domaine de la vérification formelle de circuits est très riche. Nous ne ferons pas ici une description exhaustive des travaux le concernant. La plupart du temps, il s'agit de coupler la description d'un circuit dans un certain langage avec une méthode de validation opérant sur ce langage. Cette approche de bas niveau s'appuie en général sur un prouveur de théorèmes. Citons par exemple les travaux de Purushothaman et Subrahmanyam avec le prouveur de Boyer et Moore [59] ou de Gordon avec le système HOL [35].

Nous nous intéressons ici plutôt au problème de la vérification de la spécification elle-même. Plusieurs approches permettent de passer d'une spécification haut-niveau à une description de circuit. Le formalisme Ruby [42] en est un exemple. Le langage ALPHA en est un autre. Des transformations par réécritures sont au cœur du système ALPHA tel qu'il est actuellement implanté. Celles-ci permettent d'utiliser le même langage le long de toute une chaîne de transformations, à la fois pour décrire des spécifications haut-niveau et des implantations bas-niveau. De par sa nature, proche des langages fonctionnels, ce langage est donc particulièrement adapté à des méthodes de validation de type interne. Cependant, il apparaît que des méthodes de validation de type externe compléteraient utilement le système existant. En effet, le système actuel repose sur l'hypothèse que la spécification haut-niveau initiale est correcte. Dans le cas où ces spécifications représentent des applications importantes, cette hypothèse devient de plus en plus difficile à assurer. Les motivations pour une approche de type externe sont plus longuement exposées dans l'introduction de la seconde partie.

5 Contributions

Nous donnons ici le contexte et les principales contributions de cette thèse.

- Le langage $\mathcal{L}$ a été défini par Luc Bougé en 1990 [2]. L'étude de sa sémantique opérationnelle a été faite par Luc Bougé et Jean-Luc Levaire en 1992 [12].

- Un premier système de preuve pour $\mathcal{L}$ a été donné par Luc Bougé et Gil Utard en 1994, en collaboration avec Yann Le Guyadec et Bernard Virot [11]. Ces mêmes auteurs ont travaillé sur le calcul des plus faibles préconditions [10].

- À partir de ces travaux, j'ai établi les premiers résultats de complétude du chapitre 4, et la méthodologie de preuve par annotations du chapitre 5, sous la direction de Luc Bougé.

- Gil Utard a dans sa thèse [68] amorcé le travail sur le second langage d'assertion, par une modification du calcul des plus faibles préconditions et de la preuve de complétude. Nous avons ensemble prolongé ce travail pour définir un nouveau langage et le système de preuve complet associé.

- En résumé, le travail effectué sur le langage $\mathcal{L}$ complète et réalise une synthèse des travaux précédents, notamment ceux de Bougé et Utard. Dans un premier temps, il s'agissait d'utiliser les travaux antérieurs, qui s'étaient arrêtés aux propriétés du calcul des plus faibles préconditions, pour mieux comprendre le « comportement » du système de preuve vis-à-vis de la propriété de complétude. Ensuite, j'ai proposé une approche plus « pratique » de ce système, sous la forme des preuves par annotations, et montré le bien-fondé de cette approche. Finalement, à partir d'une modification du calcul des plus faibles préconditions présentée dans la thèse de Utard, nous avons ensemble proposé une nouvelle approche. Cette dernière unifie les

précédentes tentatives. Cette thèse permet de comparer ces approches, selon des points de vue théoriques ou pratiques.

- Le langage ALPHA est développé par l'équipe API de l'IRISA. Le mémoire de DEA de Laurence Nédelka [54] donne une première idée de ce que pourrait être une sémantique opérationnelle. J'ai défini plus précisément une telle sémantique et donné ses propriétés, puis donné un cadre formel de vérification et une méthodologie associée, le tout sous la direction de Luc Bougé.

6 Publications

Nous donnons ici la liste des publications rédigées au cours de ce travail de thèse.

- Cachera (D.). – *Étude de systèmes de preuve pour les programmes data-parallèles.* – Rapport de DEA, ENS Lyon, France, 1994.
- Cachera (D.). – Deux résultats de complétude pour un langage data-parallèle simple. *In: Proc. RenPar'7*, éd. par Dekeyser, Libert et Manneback. – Mons, Belgique, mai 1995.
- Cachera (D.). – *Two completeness results about a proof system for a simple data-parallel language (full version).* – Rapport de recherche n° 95–29, ENS Lyon, LIP, décembre 1995.
- Bougé (L.) et Cachera (D.). – *On the completeness of a proof system for a simple data-parallel language.* – Rapport de recherche n° 94–42, ENS Lyon, LIP, décembre 1994.
- Bougé (L.) et Cachera (D.). – On the completeness of a proof system for a simple data-parallel language. *In: Proc. 1st EuroPar Conf.* – Stockholm, Sweden, août 1995. LNCS 966.
- Bougé (L.) et Cachera (D.). – *Proofs by annotations for a simple data-parallel language.* – Rapport de recherche n° 95–08, ENS Lyon, LIP, mars 1995.
- Bougé (L.) et Cachera (D.). – Proving data-parallel programs correct: The proof outlines approach. *In: Proc. Massively Parallel Programming Models.* – Berlin, Allemagne, octobre 1995. IEEE.
- Bougé (L.), Cachera (D.), Le Guyadec (Y.), Utard (G.) et Virot (B.). – Formal validation of data-parallel programs: introducing the assertional approach. *In: The data-parallel programming model*, éd. par G.-R. Perrin (A. Darte), pp. 252–281. – Springer Verlag, 1996.
- Bougé (L.), Cachera (D.), Le Guyadec (Y.), Utard (G.) et Virot (B.). – *Formal validation of data-parallel programs: a two-component assertional proof system for a simple language.* – Rapport de recherche n° 3033, Grenoble, INRIA, novembre 1996.
- Bougé (L.), Cachera (D.), Guyadec (Y. Le), Utard (G.) et Virot (B.). – Formal validation of data-parallel programs: a two-component assertional proof system for a simple language. *Theoretical Computer Science B*, 1997. – À paraître.
- Cachera (D.) et Utard (G.). – *Proving data-parallel programs: a unifying approach.* – Rapport de recherche n° 3032, Grenoble, INRIA, novembre 1996.
- Cachera (D.) et Utard (G.). – Proving data-parallel programs: a unifying approach. *Parallel Processing Letters*, vol. 6, n° 4, 1996, pp. 491–505.

- Bougé (L.) et Cachera (D.). – *A logical framework to prove properties of Alpha programs.* – Rapport de recherche n° 3031, Grenoble, INRIA, novembre 1996.
- Bougé (L.) et Cachera (D.). – *A logical framework to prove properties of Alpha programs (revised version).* – Rapport de recherche n° 3177, Grenoble, INRIA, juin 1997.
- Bougé (L.) et Cachera (D.). – A logical framework to prove properties of Alpha programs. *In: Proc. Application-Specific Arrays and Processors*, pp. 187–198. – Zürich, Suisse, juillet 1997. IEEE.
- Cachera (D.). – Prouver des programmes Alpha: l'approche externe. *In: Proc. RenPar'9*, éd. par Schiper (A.) et Trystram (D.). – École Polytechnique Fédérale de Lausanne, Suisse, mai 1997.

Première partie

Le langage $\mathcal{L}$

chapitre 1

Le langage $\mathcal{L}$ [1]

Dans le modèle de programmation data-parallèle, les objets de base sont des tableaux à accès parallèle. La notion de tableau est ici prise au sens large et désigne un objet multidimentionnel, quelle que soit la façon dont sont décrites ses bornes. Des actions de deux types peuvent être appliquées à ces objets : soit des opérations agissant en parallèle sur chaque composante de l'objet considéré, soit des actions de communication, ou réarrangements.

Le langage $\mathcal{L}$ a été conçu par Bougé [2] comme noyau commun à plusieurs langages data-parallèles existants. Le but était de construire un langage à la fois suffisamment simple pour pouvoir en étudier la sémantique de façon complète et précise, et suffisamment expressif pour pouvoir décrire le comportement de langages data-parallèles réels comme C*, MPL ou HyperC. La question de l'expressivité du langage $\mathcal{L}$ a été traitée par Levaire dans sa thèse [50]. Il y définit une sémantique opérationnelle qui lui permet de comparer formellement $\mathcal{L}$ avec certains des langages dont il est inspiré, et également de valider certains schémas de compilation. Des travaux ultérieurs ont permis de définir une sémantique axiomatique [10]. Notre travail sur $\mathcal{L}$ s'inscrit directement dans ce cadre.

Ce chapitre est une description rapide du langage $\mathcal{L}$ tel que nous l'utiliserons dans la suite. Nous en donnons tout d'abord une description informelle, puis une sémantique dénotationnelle.

1.1 Description informelle

Dans le langage $\mathcal{L}$, les objets de base sont des tableaux parallèles, que nous appelons *vecteurs*. Lors de l'exécution d'une action, seule une partie des indices du vecteur considéré est concernée par cette action. Les indices auxquels l'action est appliquée sont dits actifs. À un instant donné, l'ensemble des indices actifs est appelé *contexte d'activité*. On peut le définir comme un vecteur de booléens où *true* dénote l'activité et *false* l'inactivité.

Les langages data-parallèles usuels qui ont servi de base à la définition de $\mathcal{L}$ sont de nature déterministe, même si l'ordre d'exécution des instructions parallèles peut lui être non-déterministe et dépendre de l'implantation choisie. Les seuls cas de non déterminisme pouvant se produire viennent de communications de type send (par exemple en C*) ou de certains problèmes de passage de paramètres sur des indices inactifs. Mais ces cas ne correspondent pas vraiment à la « philosophie » des langages étudiés et sont ici considérés davantage comme des effets de bord. Le langage $\mathcal{L}$ ne s'attarde pas à ce niveau de détail et, dans un souci de généralité et de simplicité, il ne permet de

1. La matière de ce chapitre est tirée de *Control structures for data-parallel SIMD languages: semantics and implementation* (L. Bougé et J.-L. Levaire, [12]) et *Formal validation of data-parallel programs: a two-component assertional proof system for a simple language* (L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard et B. Virot, [5]).

décrire que des programmes déterministes. Dans le même souci de simplicité, nous ne considérons pas la partie scalaire de ces langages, principalement inspirée du langage C. On pourra considérer que les variables scalaires sont dupliquées de façon à avoir la même valeur recopiée sur tout un vecteur. Enfin, nous ne considérons que des valeurs entières ou booléennes.

Nous considérons que tous les objets de $\mathcal{L}$ sont de même dimension et sont définis sur le même domaine. Il n'y a qu'un seul type géométrique d'objet. Cette conception est semblable à celle du langage MPL, où la géométrie commune à toutes les variables de type plural est celle de l'architecture sous-jacente. Des géométries variables, commes celles décrites par les shape en C* ou les collection en HyperC, pourraient être envisagées aisément, mais au prix de notations plus complexes. Nous supposons donc ici que nous avons un unique domaine d'indices $\mathcal{D}$, sur lequel des opérations géométriques telles que décalages, rotations, etc. peuvent être définies. Par exemple, en MPL $\mathcal{D}$ serait un domaine carré $[0..1023] \times [0..1023]$, et les opérations géométriques seraient des translations toroïdales dans la direction des axes. Nous ne précisons pas systématiquement la struture du domaine $\mathcal{D}$, ni les opérations géométriques. Les exemples que nous traiterons utiliseront des domaines mono- ou bidimensionnels. Le seul point important est que nous supposons qu'il existe deux fonctions de conversion :

- $x = \mathsf{itos}(u)$ fait correspondre une valeur scalaire x à un indice u (*index to scalar*) ;
- $u = \mathsf{stoi}(x)$ fait correspondre un indice u à une valeur scalaire x (*scalar to index*).

La fonction itos est semblable à la fonction pcoord de C*, ou à la fonction iproc de MPL. Ces deux fonctions doivent respecter une seule condition : la composition ($\mathsf{stoi} \circ \mathsf{itos}$) doit être l'identité sur les indices ($\forall u \in \mathcal{D} : u = \mathsf{stoi}(\mathsf{itos}(u))$). Si le domaine $\mathcal{D}$ est $[0..N-1]$ par exemple, la fonction itos sera l'injection canonique dans les entiers, et stoi sera la fonction *modulo*N.

Par convention, les variables ou expressions vectorielles seront notées avec une lettre majuscule ($X, Y, E, \ldots$), et les indices avec une lettre minuscule ($u, v, \ldots$). La composante d'une variable X à l'indice u est notée $X|_u$.

Expressions Une *expression vectorielle* E est définie récursivement de la façon suivante. Elle peut être :

- une variable vectorielle X ;
- une constante vectorielle entière ou booléenne. Par exemple, 1 dénote le vecteur dont toutes les composantes valent l'entier 1, *True* et *False* dénotent les vecteurs dont toutes les composantes sont respectivement *true* ou *false*. Nous introduisons une constante spéciale, *This*, qui dénote le vecteur tel que pour tout u, $This|_u = \mathsf{itos}(u)$. Cette constante est l'équivalent du iproc de MPL ou du «.» de C* ;
- l'application d'un opérateur scalaire à des arguments vectoriels. Cette application se fait composante par composante. Par exemple $X + Y$ dénote le vecteur dont la composante à un indice u vaut $X|_u + Y|_u$. On remarquera que $X = Y$ définit un vecteur de booléens et non un unique booléen qui dénoterait l'égalité globale des deux expressions ;
- une *expression conditionnelle* qui correspond à la généralisation de l'expression conditionnelle du langage C. $(C?E : F)$ dénote un vecteur dont la composante à l'indice u est la composante de E à l'indice u si la composante de C à l'indice u est vraie, et la composante de F à l'indice u sinon ;

- une expression indexée $E|_A$. Une telle expression correspond à une opération de communication de type *get* (ou *fetch*) : l'expression A est évaluée, puis l'expression E ; le résultat est ensuite réarrangé de telle façon que la valeur de l'expression indexée à un indice u donné soit celle de l'expression E à l'indice $\mathsf{stoi}(A|_u)$: $(E|_A)|_u = E|_{\mathsf{stoi}(A|_u)}$. En MPL, une telle opération est notée router[A].E, en HYPERC, elle est notée [A]E.

Par exemple, l'expression suivante pourra être utilisée dans un programme calculant une convolution sur une fenêtre de largeur 3

$$(2 * X + X|_{This+1} + X|_{This-1})/4$$

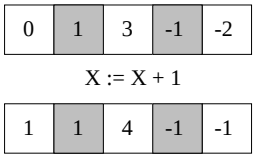
Cette expression utilise en fait un abus de notation : l'expression correcte serait $X|_{\mathsf{stoi}(This+1)}$. Il est également possible de considérer qu'une opération « + » a été définie sur les indices. Le « + » de $This + 1$ n'est alors pas de la même nature que celui de $X + \ldots$

On remarquera par ailleurs que l'évaluation des expressions ne produit pas d'effet de bord.

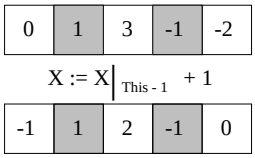
Instructions Le jeu d'instructions de $\mathcal{L}$ est le suivant.

- Affectation : $X := E$

  L'expression E est évaluée en chacun des indices actifs. La valeur obtenue est affectée à chacune des composantes actives de la variable X. L'exemple suivant consiste en une affectation par une expression n'induisant pas de communication. Le tableau au-dessus de l'instruction représente les valeurs de X avant exécution, celui en-dessous après exécution. Les indices *inactifs* sont représentés en grisé.



  Dans le cas d'une expression induisant une communication, c'est-à-dire si E contient une sous-expression indicée, il peut y avoir éventuellement évaluation d'une partie de l'expression sur un indice qui était inactif, comme l'illustre l'exemple suivant [2].



- Composition séquentielle : $S\,;\,T$

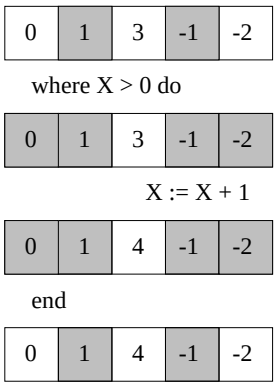
  L'exécution de T commence quand toutes les actions de S sont terminées. Le contexte d'exécution de T est le même que celui de S.

- Conditionnement : where B do S end

  Le contexte d'activité est manipulé de façon explicite dans $\mathcal{L}$. C'est le mécanisme de conditionnement qui permet cette manipulation. L'ensemble des indices actifs où l'expression vectorielle

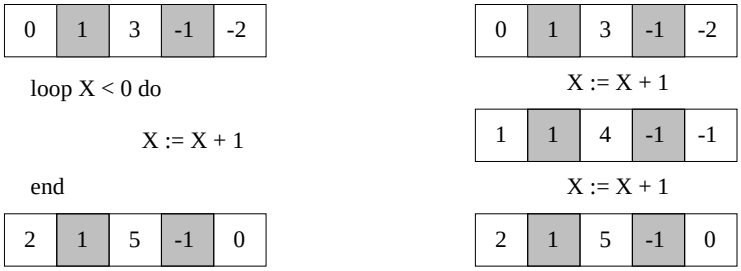
2. Dans cet exemple, nous considérons que la fonction d'indexage est cyclique : l'élément immédiatement à gauche de l'élément d'indice 0 est celui d'indice 4.

booléenne B est évaluée à faux est rendu inactif pendant toute l'exécution de S, alors que les indices qui étaient déjà inactifs restent inactifs. En sortie de cette instruction, le contexte est restauré à sa valeur initiale. Dans l'exemple suivant, X est incrémenté seulement là où il était déjà positif et où l'indice était actif.



- Itération : loop B do S end

  Les actions de S sont répétées dans le contexte courant jusqu'à ce que tous les indices actifs évaluent l'expression booléenne B à faux. On remarquera que le contexte n'est pas modifié par cette structure de contrôle, contrairement à ce qui se passe avec le while parallèle de MPL ou le whilesomewhere de C*. La forme que nous donnons, plus générale, permet de simuler ces constructions par un where imbriqué dans une boucle. Dans l'exemple suivant, la colonne de droite donne le détail des itérations.



Le langage $\mathcal{L}$ est très simple, mais suffisamment expressif pour écrire la plupart des algorithmes data-parallèles existants. Nous donnons ici deux exemples simples. Le premier exemple est celui du calcul d'une somme préfixe, suivant une méthode logarithmique classique [39]. Le domaine utilisé est $\mathcal{D} = [1..N]$. Initialement, la composante à l'indice u possède une valeur $V|_u$, et nous calculons S tel que $\forall u : S|_u = \sum_{k=1}^{k=u} V|_k$. Le programme en $\mathcal{L}$ est donné en figure 1.1. La colonne de droite est une trace d'exécution montrant les valeurs successives prises par S durant le calcul, dans le cas particulier où V vaut 1 partout. Les flèches dénotent les communications.

Le second exemple est un programme qui calcule la distance des points d'une image au fond, dans une image binaire. Le texte du programme et une trace d'exécution sont donnés en figure 1.2. Les points de l'image ont pour valeur 1 et les points du fond 0. Dans un but de simplicité, nous nous sommes restreints à une dimension, mais le programme pourrait être étendu à deux dimensions sans difficulté. Pour chaque pixel de l'image, le programme calcule le nombre de pixels qui le sépare du plus proche pixel de fond. À chaque itération, la valeur précédente de X est stockée temporairement

```
S := V; I := 1;
loop I < N do
    where This > I do
        S := S + S|_{This-I}
    end;
    I := 2 * I
end
```

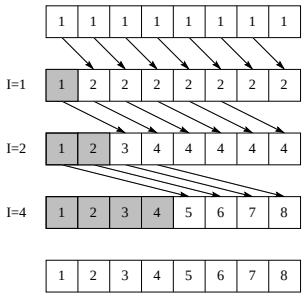


FIG. 1.1 – *Somme préfixe.*

dans la variable Y. Chaque pixel actif calcule une nouvelle distance d'après ses deux voisins. Quand le calcul est terminé, il devient inactif pour les itérations suivantes.

```
X' := 0;
loop (X' ≠ X) do
  where (X' ≠ X) do
    X' := X;
    X := Min(X|_{This-1}, X|_{This+1}) + 1;
  end
end
```

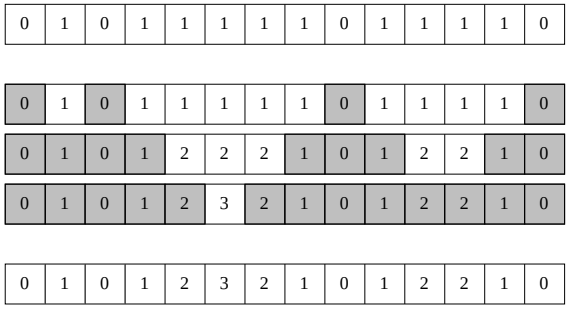


FIG. 1.2 – *Calcul de distance au fond*

1.2 Sémantique dénotationnelle

Nous donnons maintenant une sémantique dénotationnelle pour le langage $\mathcal{L}$, telle que décrite dans [5]. Cette sémantique nous permet de décrire formellement le comportement des structures de contrôle expliqué dans la section précédente. Cette description formelle nous servira de base pour toute la suite. Toutes les propriétés que nous montrerons reposeront sur le comportement décrit par cette sémantique.

La sémantique dénotationnelle décrit la fonction calculée par un programme $\mathcal{L}$, c'est-à-dire l'état des variables en sortie en fonction de l'état en entrée. Plus formellement, nous donnons les définitions suivantes.

- Un *environnement* σ est une application de l'ensemble des variables vers l'ensemble des valeurs vectorielles. L'ensemble des environnements sera noté Env. Les environnements sont étendus aux expressions de manière usuelle : $\sigma(E)$ dénote la valeur (vectorielle) obtenue en évaluant l'expression E dans l'environnement σ . Nous ne donnons pas une description exhaustive de la sémantique des expressions, mais seulement les règles les plus intéressantes :
  - $\sigma(This)|_u = \mathsf{itos}(u)$
  - $\sigma(E+F)|_u = \sigma(E)|_u + \sigma(F)|_u$
  - $\sigma(C?E:F)|_u = \begin{cases} \sigma(E)|_u & \text{si } \sigma(C)|_u = true \\ \sigma(F)|_u & \text{sinon} \end{cases}$

– $\sigma(E|_A)|_u = \sigma(E)|_{\mathsf{stoi}(\sigma(A)|_u)}$

Nous introduisons une notation de mise à jour des environnements :

$$\sigma[X \leftarrow V]$$

où X est une variable vectorielle et V un vecteur de valeurs, est l'environnement σ dans lequel la valeur de X est remplacée par V : pour toute variable Y, $\sigma[X \leftarrow V](Y) = \sigma(Y)$ si $Y \not\equiv X$, et $\sigma[X \leftarrow V](Y) = V$ si $Y \equiv X$.

- Un *contexte* c est un vecteur booléen. Il spécifie l'activité à chaque indice. Pour des raisons de lisibilité, nous utiliserons parfois le prédicat $actif_c$: $actif_c(u) \equiv c|_u = true$.
- Un *état* s est une paire (σ, c) constituée d'un environnement et d'un contexte. L'ensemble des états est muni d'une structure inductive partiellement ordonnée (*cpo*). Une telle structure présente en particulier la propriété suivante : toute chaîne (non vide) croissante d'éléments admet une borne supérieure. C'est cette propriété qui permet de justifier la bonne définition de la sémantique dénotationnelle dans le cas des boucles. Dans notre cas précis, la structure de cpo est obtenue classiquement par adjonction d'un élément spécial, $\perp$, qui dénote la non-terminaison, et par définition d'un ordre plat : $\perp \leq s$ pour tout s et deux états s et s' sont incomparables s'ils sont différents. Le cpo des états est noté *Etats*.

La sémantique $[\![S]\!]$ d'un programme S est une fonction *stricte* de *Etats* dans *Etats* : elle vérifie $[\![S]\!](\perp) = \perp$. Cette fonction est étendue aux ensembles d'états de façon usuelle. Nous donnons ci-dessous l'ensemble des règles concernant les commandes de $\mathcal{L}$. Nous ne justifierons pas ici formellement la bonne définition de ces règles, les arguments de monotonie et de continuité des fonctions définies par celles-ci étant identiques au cas impératif scalaire usuel [70].

Affectation. À chaque indice actif, la composante de la variable parallèle est mise à jour avec la valeur correspondante de l'expression

$$[\![X := E]\!](\sigma, c) = (\sigma', c),$$

avec $\sigma' = \sigma[X \leftarrow V]$ où $V|_u = \sigma(E)|_u$ si $actif_c(u)$, et $V|_u = \sigma(X)|_u$ sinon. Le contexte d'activité est préservé.

Séquencement. La composition séquentielle est simplement la composition des fonctions

$$[\![S \,;\, T]\!](\sigma, c) = [\![T]\!]([\![S]\!](\sigma, c)).$$

Itération. Le calcul de la boucle se fait de façon classique. Il se termine lorsque l'expression booléenne B s'évalue à faux pour tous les indices actifs. Nous en donnons la définition formelle suivante, qui reprend mot pour mot la méthode classique (voir par exemple [70]). Soit $\phi_k(\sigma, c)$ l'état final de l'instruction $\mathsf{loop}\ B\ \mathsf{do}\ S\ \mathsf{end}$ après avoir évalué au plus k fois le test de la boucle pour $k \geq 1$, et $\phi_0(\sigma, c) = \perp$. On a

$$\phi_{k+1}(\sigma, c) = \begin{cases} \phi_k([\![S]\!](\sigma, c)) \text{ si } \exists u : (actif_c(u) \wedge \sigma(B)|_u) \\ (\sigma, c) \quad \text{sinon} \end{cases}$$

Il est facile de montrer que si $\phi_k(\sigma, c) \neq \perp$, alors $\phi_{k+1}(\sigma, c) = \phi_k(\sigma, c)$. On peut alors définir

$$[\![\mathsf{loop}\ B\ \mathsf{do}\ S\ \mathsf{end}]\!](\sigma, c) = \bigsqcup_k \phi_k(\sigma, c)$$

où $\bigsqcup_k$ dénote la borne supérieure pour l'ordre plat sur les états. La sémantique de la boucle est alors bien définie et vérifie la relation suivante

$$[\![\textsf{loop } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = \begin{cases} [\![\textsf{loop } B \textsf{ do } S \textsf{ end}]\!]([\![S]\!](\sigma, c)) \\ \qquad\qquad \text{si } \exists u : (active_c(u) \wedge \sigma(B)|_u) \\ (\sigma, c) \text{ sinon} \end{cases}$$

Si l'évaluation diverge, on a

$$[\![\textsf{loop } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = \bot.$$

Conditionnement. La fonction calculée par une structure de conditionnement est la même que celle calculée par le corps du **where**, mais avec masquage du contexte par l'expression booléenne

$$[\![\textsf{where } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = (\sigma', c)$$

si $[\![S]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c')$, et

$$[\![\textsf{where } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = \bot$$

si $[\![S]\!](\sigma, c \wedge \sigma(B)) = \bot$. On remarquera que, dans le cas de convergence, la valeur de c', contexte en sortie de S, est ignorée. Le mécanisme de conditionnement est un mécanisme de pile, qui restaure l'ancien contexte (c) en sortie du **where**, après avoir « dépilé » le contexte intérieur au **where**.

Remarques

- Dans cette version du langage, le contexte est conservé par toute exécution qui termine : si $[\![S]\!](\sigma, c) = (\sigma', c')$, alors $c = c'$. Ce n'est plus vrai lorsque le langage est étendu avec des structures de contrôle d'échappement parallèle de type **break** telles que définies par Bougé et Levaire [12].
- On remarquera enfin que toutes les constructions du langage $\mathcal{L}$ sont déterministes.

chapitre 2

Un système de preuve par assertions en deux parties[1]

Notre but est de prouver la correction de programmes $\mathcal{L}$ en utilisant une méthode la plus proche possible de celles proposées usuellement pour les langages impératifs scalaires. Dans une méthode du type « logique de Hoare », une spécification de programme est une formule $\{P\}\ S\ \{Q\}$, où S est le programme et P et Q sont deux assertions sur les variables de S. Une telle formule signifie : si P est vraie dans un état initial et si S est exécuté à partir de cet état initial, alors si S atteint un état final, Q sera vrai dans cet état final. Nous ne nous préoccupons ici que de correction partielle, c'est-à-dire que nous ne cherchons pas à montrer que l'état final est effectivement atteint.

Établir la correction d'une spécification se fait en dérivant la formule dans un système de preuve constitué d'axiomes pour les instructions simples et de règles pour les structures de contrôle. La preuve se dérive par induction sur la syntaxe du programme. Cependant, comprendre une preuve formelle ainsi établie est difficile. Au lieu de parcourir un arbre de preuve, il est plus simple de profiter du fait que les preuves sont dirigées par la syntaxe pour les inclure dans le texte du programme. C'est la méthode proposée par Floyd [31] pour les langages impératifs scalaires (ALGOL en l'occurrence), dont la logique de Hoare est une abstraction. La preuve consiste en une insertion des assertions, appelées alors *annotations* dans le texte du programme. Lorsque le contrôle atteint un point où est insérée une annotation, l'assertion correspondante doit être vraie dans l'état courant.

En ce qui concerne $\mathcal{L}$, nous cherchons donc un cadre logique adapté, c'est-à-dire

- un langage d'assertion approprié pour le cas data-parallèle ;
- la possibilité de réaliser des preuves par assertions ;
- un ensemble de règles suffisamment riche pour établir les spécifications voulues.

Nous montrons dans les chapitres 2 à 5 comment le système de preuve proposé initialement par Bougé et al. [11] répond à ces critères.

2.1 Langage d'assertion

De même que pour les langages scalaires classiques, le langage d'assertion doit nous permettre d'exprimer les propriétés voulues sur les variables du programme. Il doit donc au moins « contenir »

1. La matière de ce chapitre est tirée de *Formal validation of data-parallel programs: introducing the assertional approach* (L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard et B. Virot, [4]).

les expressions arithmétiques et booléennes de $\mathcal{L}$. Mais ce n'est pas suffisant. En effet, un principe très important qui prévaut lors de la définition d'une sémantique est celui de *compositionnalité* : la sémantique d'un ensemble doit être fonction de la sémantique de ses parties. Dans le cas de la logique de Hoare, cela se traduit par le fait que l'on peut remplacer un programme par un autre programme acceptant la même spécification, c'est-à-dire les mêmes pré- et postconditions. Si l'on prend le cas d'une composition séquentielle par exemple, on peut prouver

$$\{P\}\ S;T\ \{Q\}$$

en trouvant une assertion R telle que l'on puisse prouver

$$\{P\}\ S\ \{R\} \text{ et } \{R\}\ T\ \{Q\}.$$

Si T' est un programme tel que

$$\{R\}\ T'\ \{Q\}$$

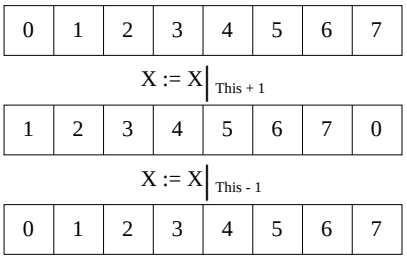
alors on aura également

$$\{P\}\ S;T'\ \{Q\}.$$

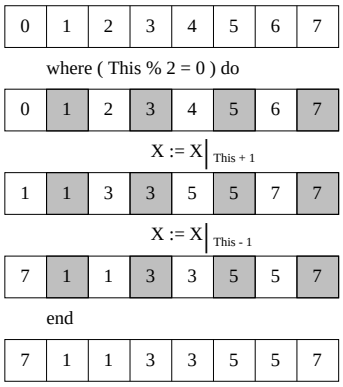
Voyons ce qui se passe dans le cas data-parallèle. Considérons par exemple le programme S suivant

$$X := X|_{This+1}; X := X|_{This-1}$$

Dans le cas où le contexte initial est actif sur tout les indices, l'exécution donne



Si P est un prédicat quelconque portant sur la variable X, on devrait pouvoir prouver $\{P\}\ S\ \{P\}$. On considère donc que S est équivalent au programme $X := X$. Considérons maintenant une exécution à partir d'un contexte où seuls les indices pairs sont actifs. On obtient



Il n'est alors plus question de remplacer S par $X := X$. Ceci montre que le langage d'assertion doit tenir compte du contexte si l'on veut pouvoir disposer d'une sémantique compositionnelle.

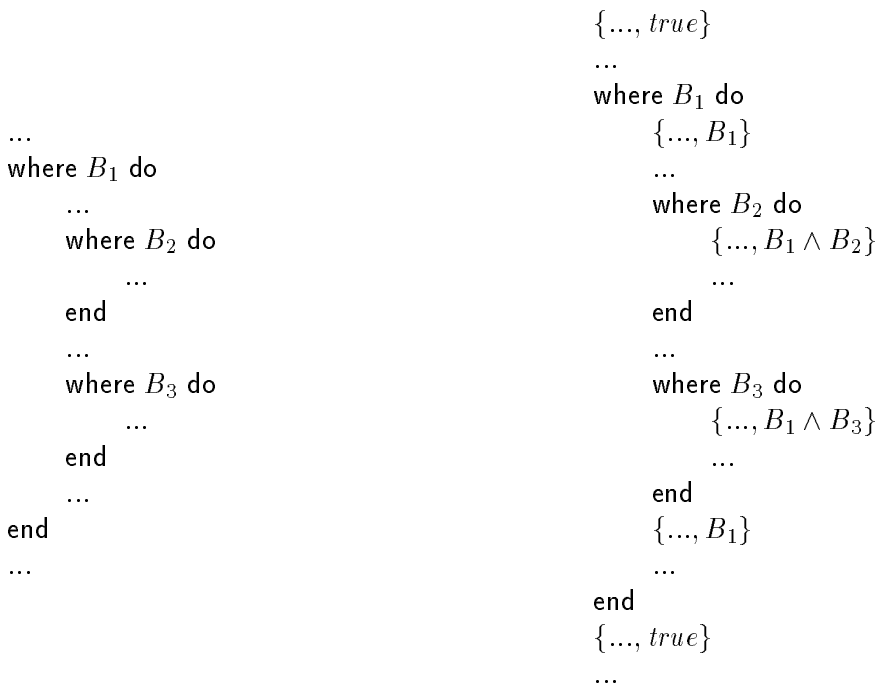


FIG. 2.1 – *Schéma de preuve par annotations*

Il existe plusieurs façons de prendre en compte le contexte dans les assertions. L'une d'elles, introduite par Narayana et Clint [20], consiste à réserver une variable spéciale (notée $\sharp$), qui dénote toujours le contexte courant. Cette méthode a été développée par Le Guyadec et Virot pour le langage $\mathcal{L}$ [47]. Nous développons ici une autre approche, historiquement antérieure en ce qui concerne $\mathcal{L}$. Cette approche traduit de façon immédiate le mécanisme de masquage du **where** dans le langage d'assertion. En effet, le contexte y est représenté par une expression vectorielle booléenne. C'est en effet la solution la plus naturelle, puisque le mécanisme de masquage de contexte introduit par la structure de conditionnement utilise une expression vectorielle booléenne. À un instant donné (en supposant que la valeur des variables n'ait pas changé entre temps), le contexte correspond à la conjonction des évaluations des expressions booléennes de toutes les structures de conditionnement englobantes, ces évaluations se faisant au moment de rentrer dans le corps du **where**. L'expression booléenne de contexte de l'assertion sera donc la conjonction des expressions booléennes des structures de conditionnement englobantes.

Cette approche a l'avantage de la simplicité. Nous remarquerons dans ce chapitre qu'une autre de ses qualités est d'être utilisable pour définir une méthode de preuve par annotations. Considérons par exemple la figure 2.1, qui montre un schéma de programme $\mathcal{L}$ et une esquisse de preuve par annotations.

L'assertion $\{..., true\}$ signifie que tous les indices sont actifs au début de l'exécution. L'assertion $\{..., B_1 \wedge B_2\}$ signifie que les indices actifs sont les u tels que $B_1|_u \wedge B_2|_u$ est vrai. Nous verrons plus loin dans quelles conditions une preuve par annotations de ce type est valide.

Dans un premier temps, nous travaillerons donc avec le langage des assertions en deux parties introduit par Bougé et al. dans [11], et avec le système de preuve reposant sur ce langage. Nous donnons la définition suivante.

Définition 1 (assertion) *Une assertion $\{P, C\}$ est une paire composée d'un prédicat P sur les variables de programme et d'une expression booléenne de contexte C.*

Prédicats Il nous reste à expliciter le langage utilisé pour exprimer les prédicats.

- Une *expression d'indice* est une variable d'indice ($u, v, etc.$), une constante ($0, 1, etc.$ par exemple si l'on considère un domaine monodimensionnel indexé par les entiers), une combinaison d'expressions d'indice par un opérateur géométrique ($u + 1$ par exemple pour un décalage sur un domaine monodimensionnel), ou la fonction stoi appliquée à une expression scalaire.
- Une *expression scalaire* est une constante scalaire ($0, true, false, etc.$), une variable scalaire ($x, y, etc.$), la combinaison d'expressions scalaires avec un opérateur scalaire, une expression vectorielle du langage de programmation indicée par une expression d'indice $E|_u$, ou la fonction itos appliquée à une expression d'indice.
- Une formule est une expression scalaire de type booléen, une combinaison de formules par des opérateurs logiques, ou une formule quantifiée sur une variable scalaire ou une variable d'indice (dans ce dernier cas, la quantification se fait implicitement sur le domaine spatial).
- Un prédicat est une formule close pour les variables scalaires et d'indice.

Voici quelques exemples de prédicats.

$\forall u : X\|_u = 0$	Toutes les composantes de X ont pour valeur 0
$\forall u : \forall v : X\|_u = X\|_v$	Toutes les composantes de X sont égales
$\forall u : X\|_u = Y\|_u$	X et Y ont la même valeur à chaque indice
$\forall u : impair(u) \Rightarrow (X\|_{u+1} = X\|_u)$	À tous les indices impairs, la composante de droite est la même que la composante locale
$\exists i : \forall u : X\|_u = i$	X est un vecteur constant

On remarquera qu'il n'y a pas de quantification sur les variables vectorielles. On remarquera également que $X = Y$ n'est pas un prédicat. C'est un vecteur booléen qui contient à chaque composante l'égalité des composantes correspondantes de X et Y. Le prédicat d'égalité globale des vecteurs X et Y est $\forall u : X|_u = Y|_u$, et sera parfois noté $X \equiv Y$ dans la suite.

Satisfiabilité Un prédicat est une formule close par rapport aux variables scalaires et d'indice. Il est donc possible de définir sa valeur de vérité dans un environnement. Nous ne détaillerons pas ici les règles d'évaluation d'un prédicat dans un environnement, qui sont tout à fait standard et intuitivement claires. Remarquons seulement que les variables scalaires sont entières ou booléennes, et que les variables d'indice parcourent le domaine $\mathcal{D}$. On note $\sigma \models P$ le fait que le prédicat P est vrai dans l'environnement σ. Si le prédicat P est vrai dans tout environnement, nous écrivons $\models P$, ou même P si le contexte est clair. Nous pouvons maintenant définir la notion de validité d'une assertion dans un état.

Définition 2 (Satisfiabilité) *Soit (σ, c) un état, et $\{P, C\}$ une assertion. On dit que l'état (σ, c)* satisfait *l'assertion $\{P, C\}$, ce que l'on note $(\sigma, c) \models \{P, C\}$, si $\sigma \models P$ et $\sigma(C) = c$. On dira également que l'assertion $\{P, C\}$ est* valide *dans l'état (σ, c).*

Par convention, $\perp$ satisfait toute assertion.
L'ensemble des états satisfaisant une assertion $\{P, C\}$ est noté $[\![\{P, C\}]\!]$.

Implication Une notion fondamentale et indispensable pour pouvoir manipuler des assertions dans un système de preuve est celle d'implication des assertions. Il nous faut redéfinir cette notion pour prendre en compte la partie contexte.

Définition 3 (Implication) *Soit $\{P, C\}$ et $\{Q, D\}$ deux assertions. On dit que $\{P, C\}$ implique $\{Q, D\}$, que l'on note*

$$\{P, C\} \Rightarrow \{Q, D\}$$

si et seulement si

$$\models P \Rightarrow Q \text{ et } \models (P \Rightarrow (\forall u : C|_u = D|_u))$$

Cette définition respecte l'interprétation ensembliste usuelle de l'implication exprimée par la propriété suivante.

Proposition 1 *Soit $\{P, C\}$ et $\{Q, D\}$ deux assertions. On a*

$$\{P, C\} \Rightarrow \{Q, D\}$$

si et seulement si

$$[\![\{P, C\}]\!] \subseteq [\![\{Q, D\}]\!].$$

Nous donnons la preuve de cette propriété simple afin de bien illustrer le mécanisme de l'implication et la façon dont sont traitées les assertions.

Preuve. *Supposons $\{P, C\} \Rightarrow \{Q, D\}$. Soit un état (σ, c) dans $[\![\{P, C\}]\!]$. On a par définition $\sigma \models P$ et $\sigma(C) = c$. Par définition de l'implication, on a alors $\sigma \models Q$ et $\sigma \models (\forall u : C|_u = D|_u)$. Donc $\sigma(D) = \sigma(C) = c$. On en conclut que $(\sigma, c) \models \{Q, D\}$, donc que $(\sigma, c) \in [\![\{Q, D\}]\!]$.*

Réciproquement, supposons $[\![\{P, C\}]\!] \subseteq [\![\{Q, D\}]\!]$. Soit un environnement σ tel que $\sigma \models P$. Posons $c = \sigma(C)$. On a alors $(\sigma, c) \models \{P, C\}$. Par hypothèse, on a $(\sigma, c) \models \{Q, D\}$. Donc par définition de la satisfiabilité, $\sigma \models Q$ et $\sigma(D) = c$. On a donc $\sigma(D) = \sigma(C)$, donc $\sigma \models (\forall u : C|_u = D|_u)$. On en conclut que $\{P, C\} \Rightarrow \{Q, D\}$. ❑

Substitution Nous introduisons finalement un mécanisme de substitution des variables vectorielles dans les prédicats. Par définition de notre langage d'assertion, les variables vectorielles sont toutes libres. La substitution de la variable vectorielle X par l'expression vectorielle E dans le prédicat P est notée $P[E/X]$. Elle se définit par induction sur la structure du prédicat P. Nous ne donnons pas ici la définition formelle, l'intuition en étant suffisamment claire. Une telle définition peut être trouvée dans l'ouvrage de Apt et Olderog ([1]). La notion de substitution peut être étendue des prédicats à toute expression contenant des variables vectorielles. Voici quelques exemples de substitution pour illustrer notre propos.

P	E	$P[E/X]$
$\forall u : X\|_u = 1$	$X + 1$	$\forall u : X\|_u + 1 = 1$ $\equiv \forall u : X\|_u = 0$
$\forall u : X\|_u = 1$	$(This = 0 ? X + 1 : Y)$	$\forall u : (This\|_u = 0 ? X\|_u + 1 : Y\|_u) = 1$ $\equiv X\|_0 = 0 \wedge \forall u \neq 0 : Y\|_u = 1$

Le fait important est que le lemme de substitution habituel s'étend à nos prédicats data-parallèles [1]. Ce lemme permet de relier le mécanisme syntaxique de substitution à sa traduction sémantique.

Lemme 1 (Substitution) *Soit P un prédicat, X une variable vectorielle, et E une expression vectorielle.*

$$\sigma \models P[E/X] \qquad ssi \qquad \sigma[X \leftarrow \sigma(E)] \models P$$

Preuve. *La preuve se fait simplement par induction sur la structure des prédicats. Nous n'en donnons pas les détails techniques. Remarquons seulement que la différence par rapport à [1] tient dans le fait que nous considérons des substitutions de vecteurs X dans leur ensemble, alors que l'approche de Apt et Olderog permet de substituer des composantes de vecteurs $X[i]$. Par exemple, il est possible de substituer la composante $X[X[i]]$ par une expression. Il faut alors considérer différents cas, suivant que $X[i] = i$ ou non. C'est d'ailleurs aussi le problème de l'approche de Stewart et Wray. La notion de substitution que nous proposons ici, qui est une notion globale et ne permet pas d'utiliser des variables vectorielles dans les expressions d'indice, est plus simple.* ❑

2.2 Système de preuve

Validité Nous avons maintenant un langage d'assertion à notre disposition. Il nous est donc possible de définir des spécifications de la forme $\{P, C\}\ S\ \{Q, D\}$. Nous devons préciser la notion de *correction* (ou *validité*) d'une spécification. Cette notion de correction repose sur la sémantique dénotationnelle définie précédemment.

Définition 4 (Validité) *Soit S un programme $\mathcal{L}$, $\{P, C\}$ et $\{Q, D\}$ deux assertions. On dit que la spécification $\{P, C\}\ S\ \{Q, D\}$ est valide, ce que l'on note $\models \{P, C\}\ S\ \{Q, D\}$, si pour tout état (σ, c),*

$$(\sigma, c) \models \{P, C\} \qquad \Rightarrow \qquad [\![S]\!](\sigma, c) \models \{Q, D\}$$

La notion de validité présentée ici concerne uniquement la correction partielle. Nous ne prenons pas en compte les problèmes de terminaison. En d'autres mots : si le programme termine, alors l'état final vérifie la postcondition [2]. La notion de correction totale est plus complexe, puisqu'il faut également prouver la terminaison. En logique de Hoare classique, ce problème est traité par l'introduction de la notion de variant. Une méthode similaire devrait être employée dans notre cas data-parallèle. Ce problème de la terminaison est orthogonal à celui de la preuve de correction et dépasse le cadre de ce travail.

Système de preuve Notre but est de trouver un ensemble d'axiomes et de règles qui nous permette de prouver la validité d'une spécification donnée. Nous utilisons ici dans un premier temps le système de preuve présenté par Bougé et al. dans [11]. Revenons un instant au cas des langages impératifs scalaires, et considérons une affectation $x := e$. L'axiome permettant de prouver une spécification faisant intervenir cette affectation est

$$\overline{\{P[e/x]\}\ x := e\ \{P\}}$$

2. Rappelons que l'état $\perp$ satisfait toute assertion, ce qui justifie l'adéquation de la définition.

Cet axiome fonctionne « en arrière »: pour que l'assertion P soit vraie après exécution de l'affectation, il faut que P dans laquelle on a substitué les occurrences libres de la variable x par l'expression e soit vraie avant l'exécution. La correction de cet axiome est directement liée au lemme de substitution 1. Une généralisation directe de cet axiome au langage $\mathcal{L}$ serait

$$\overline{\{P[(C?E:X)/X], C\}\ X := E\ \{P, C\}}$$

Cet axiome exprime le fait que l'affectation locale $X|_u := E|_u$ est effectuée uniquement pour les indices u actifs, c'est-à-dire ceux pour lesquels C est vrai. Dans ce cas, la valeur précédente de $X|_u$ est $E|_u$. Dans le cas d'un indice u inactif, la valeur de X n'est pas modifiée. Ceci correspond exactement à l'expression conditionnelle $(C?E:X)$.

Malheureusement, cette généralisation n'est pas correcte dans tous les cas. Par exemple, la spécification

$$\{true, Y = 2\}\ X := 1\ \{true, Y = 2\}$$

est valide. Cependant, celle-ci

$$\{true, X = 2\}\ X := 1\ \{true, X = 2\}$$

ne l'est pas. En effet, après l'affectation de X, le contexte qui était initialement décrit par le prédicat $X = 2$ ne peut plus être décrit par ce même prédicat, puisque la valeur de X a été modifiée. Une autre solution pour généraliser l'axiome d'affectation au cas data-parallèle pourrait être de substituer également X dans l'expression de contexte. Choisissons tout d'abord de substituer X globalement, avec la règle suivante

$$\overline{\{P[(C?E:X)/X], C[E/X]\}\ X := E\ \{P, C\}}.$$

Cette solution n'est pas non plus satisfaisante. Prenons l'exemple suivant décrit par Utard dans sa thèse ([68]). Soit l'instruction d'affectation

$$X := X + 1$$

et la postcondition

$$\{true, X = 1\}$$

L'application du principe de substitution donnerait comme précondition

$$\{true, X = 0\}$$

Cette précondition n'est pas correcte. Considérons en effet un état (σ, c) tel que

$$(\sigma, c) \models \{true, X = 0\}$$

avec $\sigma(X)|_0 = 1$, donc $c|_0 = \mathit{false}$. Après exécution de l'affectation, on a toujours $\sigma(X)|_0 = 1$, ce qui correspond à l'expression de contexte de la postcondition exprimée à l'indice 0. En revanche on a toujours $c|_0 = \mathit{false}$ puisque le contexte n'est pas modifié. La précondition et la postcondition ne sont pas en adéquation.

Tentons d'explorer une troisième voie, plus complexe. Il s'agit maintenant de substituer X par E dans l'expression de contexte, mais seulement pour les indices actifs. Ceci nous oblige à effectuer

la substitution de façon récursive. Supposons que l'on soit capable de trouver un vecteur C' vérifiant la relation

$$C' = C[(C'?E : X)/X]$$

L'axiome pourrait alors s'écrire

$$\frac{}{\{P[(C'?E : X)/X], C'\}\ X := E\ \{P, C\}}.$$

Le problème vient de ce que C' doit être défini de façon récursive. On peut essayer de partir d'un contexte où tous les indices sont actifs et itérer l'application

$$Y \mapsto C[(Y?E : X)/X]$$

Cette méthode nous ramène à la précondition

$$\{true, X = 0\}$$

qui d'après la tentative précédente n'est pas correcte. Si en revanche on part d'un contexte où tous les indices sont inactifs, on obtient après deux substitutions la précondition

$$\{true, ((X = 1?X + 1 = 1 : x = 1)?X + 1 : X) = 1\}$$

qui malgré sa complexité est une précondition correcte. Malheureusement ce résultat n'est pas généralisable. Il nous faudrait en effet trouver un cpo sur l'ensemble des vecteurs booléens qui ait *False* comme plus petit élément, et telle que l'application à itérer soit continue sur ce cpo. Ceci semble impossible. Quoi qu'il en soit, même si une restriction sur les expressions nous permettait de généraliser cette méthode, elle serait totalement inapplicable, puisque la complexité de la substitution dépasserait de beaucoup celle de la structure des expressions, et dépendrait même des valeurs des variables libres.

Pour généraliser l'axiome d'affectation au cas data-parallèle, nous devons donc opérer certaines restrictions. Nous allons considérer uniquement le cas où les variables apparaissant dans l'expression de contexte ne sont pas modifiées par le programme. Suivant les notations de Apt et Olderog ([1]), nous donnons les définitions suivantes.

Définition 5 *Soit un programme S. L'ensemble de toutes les variables apparaissant dans S est noté $Var(S)$. Le comportement de S ne dépend que de ces variables.*

L'ensemble des variables apparaissant en partie gauche d'une affectation est noté $Change(S)$. Seules ces variables peuvent être modifiées par l'exécution de S.

Soit E une expression. L'ensemble des variables (libres) de E est noté $Var(E)$. La valeur de E ne dépend que de ces variables.

Le système de preuve que nous donnons ici fait donc l'hypothèse que les expressions de contexte ne sont pas modifiées par le programme : $Change(S) \cap Var(C) = \emptyset$. Avec cette hypothèse, nous pouvons donner la règle de l'affectation.

Règle 1 (Affectation : $X := E$) *L'axiome usuel est étendu pour prendre en compte le contexte. La restriction syntaxique sur X garantit le fait que le contexte n'est pas modifié.*

$$\frac{X \notin Var(C)}{\{P[(C?E : X)/X], C\}\ X := E\ \{P, C\}}$$

Par exemple, considérons le programme $X := 1$ et la postcondition $\{\forall u : (Y|_u = X|_u), Y = 2\}$: les vecteurs X et Y ont toutes leurs composantes égales, et les indices actifs sont ceux pour lesquels la composante de Y vaut 2. L'application de la règle donne la spécification suivante

$$\begin{array}{l}\{\forall u : (Y|_u = ((Y|_u = 2)?1 : X|_u), Y = 2\}\\ X := 1\\ \{\forall u : (Y|_u = X|_u), Y = 2\}\end{array}$$

Celle-ci peut se réduire en

$$\begin{array}{c}\{\forall u : (Y|_u = 2 \Rightarrow Y|_u = 1) \wedge (Y|_u \neq 2 \Rightarrow Y|_u = X|_u), Y = 2\}\\ X := 1\\ \{\forall u : (Y|_u = X|_u), Y = 2\}\end{array}$$

c'est-à-dire

$$\begin{array}{c}\{\forall u : (Y|_u \neq 2) \wedge (Y|_u = X|_u), Y = 2\}\\ X := 1\\ \{\forall u : (Y|_u = X|_u), Y = 2\}\end{array}$$

Les autres règles suivent naturellement :

Règle 2 (Séquencement: $S;T$) *C'est une généralisation immédiate du cas usuel*

$$\frac{\{P,C\}\ S\ \{R,C\},\ \{R,C\}\ T\ \{Q,C\}}{\{P,C\}\ S;T\ \{Q,C\}}$$

Règle 3 (Itération: loop B do S end) *Comme dans le cas scalaire, la règle utilise un invariant. Le prédicat choisi doit ici être invariant à la fois par rapport aux valeurs des variables et par rapport au contexte.*

$$\frac{\{I \wedge \exists u : (C|_u \wedge B|_u), C\}\ S\ \{I, C\}}{\{I, C\}\ \mathsf{loop}\ B\ \mathsf{do}\ S\ \mathsf{end}\ \{I \wedge \forall u : (C|_u \Rightarrow \neg B|_u), C\}}$$

Règle 4 (Conditionnement: where B do S end) *Conformément à la sémantique dénotationnelle définie dans le chapitre précédent, le contexte à l'intérieur de la structure de conditionnement est la conjonction du contexte englobant et de l'expression vectorielle booléenne de conditionnement.*

$$\frac{\{P, (C \wedge B)\}\ S\ \{Q, D\},\ \mathit{Change}(S) \cap \mathit{Var}(C) = \emptyset}{\{P,C\}\ \mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}\ \{Q,C\}}$$

On remarquera que l'expression de contexte en sortie D est ignorée, de même que l'on ignorait le contexte de sortie dans la sémantique dénotationnelle. De plus, il n'est pas nécessaire de supposer que $\mathit{Change}(S) \cap \mathit{Var}(B) = \emptyset$. Par exemple, si l'on considère la spécification valide suivante

$$\{\forall u : X|_u \geq 0, X \geq 1\}\ X := X + 1\ \{\forall u : X|_u \geq 0, X \geq 2\}$$

la correction de cette règle de conditionnement nous fournit la validité de

$$\begin{array}{l}\{\forall u : X|_u \geq 0, \mathit{True}\}\\ \mathsf{where}\ \ X \geq 1\ \ \mathsf{do}\ \ X := X + 1\ \ \mathsf{end}\\ \{\forall u : X|_u \geq 0, \mathit{True}\}\end{array}$$

La règle suivante, appelée règle de conséquence ou d'affaiblissement, nous permet d'affaiblir la postcondition et de renforcer la précondition.

Règle 5 (Conséquence) *Conformément à la définition 3, nous avons*

$$\frac{\{P,C\} \Rightarrow \{P_1,C_1\},\ \{P_1,C_1\}\ S\ \{Q_1,D_1\},\ \{Q_1,D_1\} \Rightarrow \{Q,D\}}{\{P,C\}\ S\ \{Q,D\}}$$

Rappelons que $\{P,C\} \Rightarrow \{P_1,C_1\}$ *est un raccourci pour* $\models \{P,C\} \Rightarrow \{P_1,C_1\}$. *En d'autres termes, toutes les implications* valides *sont promues au rang d'axiomes, comme en logique de Hoare classique [1].*

Notation (Prouvabilité). Si une spécification $\{P,C\}\ S\ \{Q,D\}$ est dérivable dans le système de preuve constitué des règles 1 à 5, on le note

$$\vdash \{P,C\}\ S\ \{Q,D\}$$

Correction La première propriété à vérifier pour ce système de preuve est sa *correction*. On ne doit pouvoir prouver que des spécifications valides. Nous avons en effet la proposition suivante.

Proposition 2 (Correction) *Le système de preuve* $\vdash$ *est* correct *: si*

$$\vdash \{P,C\}\ S\ \{Q,D\}$$

alors

$$\models \{P,C\}\ S\ \{Q,D\}.$$

La preuve de cette proposition se fait par induction sur la dérivation de la preuve. La vérification se fait sur les axiomes, puis sur les règles de dérivation. Nous n'en donnons pas les détails, purement techniques. Le lecteur intéressé pourra trouver une preuve de même type (pour un langage scalaire) dans l'ouvrage de Apt et Olderog ([1]).

2.3 Bilan

Nous avons donné dans ce chapitre un premier système de preuve pour le langage $\mathcal{L}$. Avant d'aller plus loin, nous allons montrer sur un exemple comment il peut être utilisé pour dériver la preuve d'une spécification. L'exemple traité permettra de plus de mettre en lumière une insuffisance de ce système. Nous verrons dans les chapitres suivants comment pallier à ce type d'insuffisances.

chapitre 3

Un exemple de preuve

Nous reprenons ici l'exemple du calcul de la distance des points d'une image au fond. Nous rappelons tout d'abord (figure 3.1) le programme ainsi que la trace d'exécution qui était donnée en exemple.

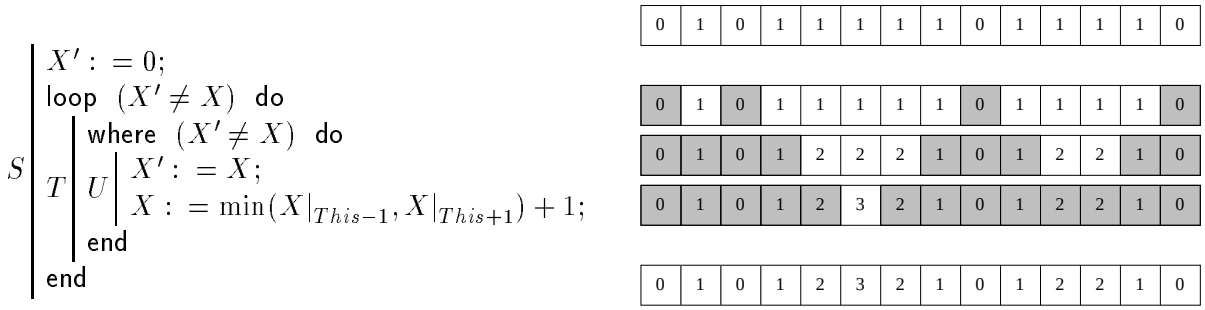


FIG. 3.1 – *Calcul de distance au fond*

Au début, l'image binaire est stockée dans X et tous les indices sont actifs. À chaque itération, chaque point de l'image qui n'a pas encore calculé sa distance finale (c'est-à-dire dont la valeur de X n'est pas stabilisée) met à jour la valeur de sa « distance provisoire » en calculant le minimum de celles de ses deux voisins. Après terminaison, chaque composante $X|_u$ contient la distance au fond. Cette distance est dénotée par $D|_u$.

Notons S le programme dans son entier, T le sous-programme de S correspondant au corps de la boucle, et U le sous-programme de T correspondant au corps du **where**. Notons B l'expression vectorielle booléenne $X' \neq X$. Nous voulons donc prouver la spécification

$$\{true, True\}\ S\ \{\forall u : X|_u = D|_u, True\}.$$

Le programme S est composé principalement d'une boucle et d'un **where** imbriqué dans cette boucle. L'arbre de preuve que nous devons construire sera donc schématiquement de la forme suivante.

$$\dfrac{\dfrac{\{I \wedge \exists u : B|_u, B\}\ U\ \{I, \ldots\}}{\{I \wedge \exists u : B|_u, True\}\ \textsf{where}\ B\ \textsf{do}\ U\ \textsf{end}\ \{I, True\}}}{\{I, True\}\ \textsf{loop}\ B\ \textsf{do where}\ B\ \textsf{do}\ U\ \textsf{end end}\ \{I \wedge \forall u : \neg B|_u, True\}}$$

Cet arbre repose sur l'utilisation des règles de l'itération et du conditionnement. Le prédicat I dénote l'invariant de boucle, qui est au cœur de la preuve.

À coté de cet « arbre principal », nous devons également montrer que l'invariant est bien conforme à la spécification. Ceci est exprimé par les deux propriétés suivantes.

$$\vdash \{true, True\}\ X' := 0\ \{I, True\}$$
$$\{I \wedge \forall u : \neg B|_u, True\} \Rightarrow \{\forall u : X|_u = D|_u, True\}$$

La première propriété sert à établir la précondition, en tenant compte de l'initialisation de X', la seconde la postcondition, en utilisant la règle de conséquence.

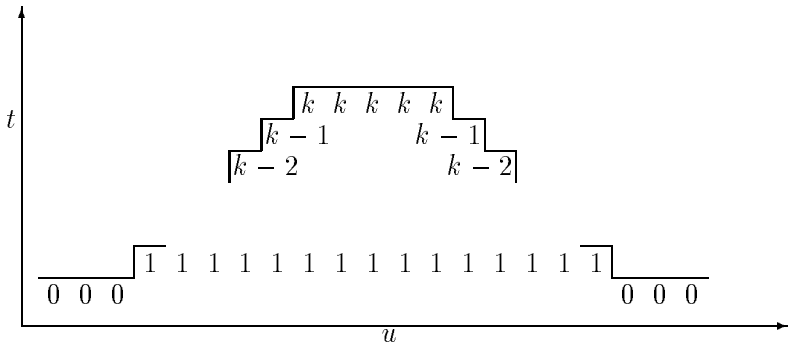


FIG. 3.2 – *Évolution de X. À l'itération k, les composants $X|_u$ tels que la distance au fond est supérieure ou égale à k sont actifs. Les autres composants sont inactifs, et leur valeur est égale à la distance. Ce fait est aussi vrai pour leurs voisins.*

Nous devons donc trouver un invariant I. Observons pour cela la figure 3.2. Nous devons exprimer le fait que les composants $X|_u$ tels que $X|_u = Y|_u$ ont déjà calculé leur distance. De plus, ce fait est également vrai pour les voisins de ce composant. Ceci se traduit par l'implication

$$X|_u = X'|_u \Rightarrow \begin{cases} X|_u = D|_u \\ X|_{u-1} = D|_{u-1} \\ X|_{u+1} = D|_{u+1} \end{cases}$$

Sinon, les composants $X|_u$ tels que $X|_u \neq X'|_u$ ont tous la même valeur, k, qui est le nombre courant d'itérations. De plus, leurs voisins qui ont déjà calculé leur distance (c'est-à-dire tels que $X|_{u\pm1} = X'|_{u\pm1}$) sont égaux à $k-1$. Ceci se traduit par l'implication

$$X|_u \neq X'|_u \Rightarrow \begin{cases} X|_u = k \wedge k \leq D|_u \\ X|_{u-1} = X'|_{u-1} \Rightarrow X|_{u-1} = k-1 \\ X|_{u+1} = X'|_{u+1} \Rightarrow X|_{u+1} = k-1 \end{cases}$$

Résumons-nous : l'invariant est de la forme

$$I \equiv \{\exists k : \forall u : J(k,u), True\}$$

avec

$$J(k,u) \equiv \begin{cases} X|_u = X'|_u \Rightarrow \begin{cases} X|_u = D|_u \\ X|_{u-1} = D|_{u-1} \\ X|_{u+1} = D|_{u+1} \end{cases} \\ X|_u \neq X'|_u \Rightarrow \begin{cases} X|_u = k \wedge k \leq D|_u \\ X|_{u-1} = X'|_{u-1} \Rightarrow X|_{u-1} = k-1 \\ X|_{u+1} = X'|_{u+1} \Rightarrow X|_{u+1} = k-1 \end{cases} \end{cases}$$

Les preuves visant à établir les pré- et postconditions (les deux propriétés ci-dessus) sont quasi immédiates et ne seront pas détaillées ici. Nous nous attacherons davantage à montrer que I est bien un invariant. Pour cela, il nous faut entrer dans le corps du where et prouver

$$\{I \wedge \exists u : X'|_u \neq X|_u, X' \neq X\}\ U\ \{I, C'\}$$

où C' est une expression de contexte quelconque. Un problème va alors se poser. En effet, les variables apparaissant dans l'expression de conditionnement sont modifiées par le programme. Il ne nous est dans ce cas pas possible de prouver directement la propriété voulue sur U, du fait des restrictions apportées à la règle d'affectation. Nous allons ici introduire de façon tout à fait informelle une méthode permettant de prouver quand même la spécification voulue. La description précise de cette méthode ainsi que sa justification seront données au chapitre suivant, consacré à la preuve de complétude.

Nous considérons pour cela une « nouvelle » variable, qui va servir à stocker temporairement le contexte à l'intérieur du corps du where. Appelons cette variable Tmp. Cette variable est nouvelle dans le sens où elle n'apparaît ni dans le programme ni dans les prédicats introduits jusqu'ici. Au lieu d'utiliser l'assertion $\{I, B\}$, nous allons utiliser $\{I, Tmp\}$. Supposons que nous ayons réussi à prouver

$$\vdash \{P, Tmp\}\ U\ \{I, Tmp\},$$

où P est un prédicat que nous expliciterons ensuite. Nous pouvons ensuite utiliser la règle de conséquence. En effet, nous avons

$$\{P \wedge Tmp = (X' \neq X), X' \neq X\} \Rightarrow \{P, Tmp\}$$

qui nous donne

$$\vdash \{P \wedge Tmp = (X' \neq X), X' \neq X\}\ S\ \{I, Tmp\}$$

Nous appliquons alors la règle du where, qui donne

$$\vdash \{P \wedge Tmp = (X' \neq X), True\}\ \textsf{where}\ \ X' \neq X\ \ \textsf{do}\ \ S\ \ \textsf{end}\ \{I, True\}$$

Maintenant que nous sommes « sortis » de la structure de conditionnement, il faut nous débarrasser de la variable temporaire qui n'apparaît ni dans l'invariant ni dans le programme. Il nous faut pour cela faire appel à une règle supplémentaire que nous allons ajouter à notre système de preuve, qui est la règle d'*élimination des variables cachées*. En effet, une telle variable ne peut avoir d'influence sur la spécification initiale. Nous ne faisons ici qu'introduire cette règle, sa justification ainsi que la notion précise de variable cachée étant données dans le chapitre suivant.

Règle 6 (Élimination des variables cachées)

$$\frac{\{P, C\}\ S\ \{Q, D\}, \quad Tmp \notin Var(S) \cup Var(Q) \cup Var(D)}{\{P[E/Tmp], C[E/Tmp]\}\ S\ \{Q, D\}},$$

où E est une expression vectorielle quelconque.

L'application de cette règle avec $E \equiv (X' \neq X)$ nous donne alors

$$\vdash \{P[(X' \neq X)/Tmp], True\}\ \textsf{where}\ \ X' \neq X\ \ \textsf{do}\ \ S\ \ \textsf{end}\ \{I, True\}$$

Il nous faut maintenant expliciter P et prouver que

$$I \wedge \exists u : X|_u \neq X'|_u \Rightarrow P[(X' \neq X)/Tmp]$$

Ceci permet d'appliquer la règle de conséquence et la règle de l'itération, et concluera la preuve.

Le prédicat P est obtenu en appliquant deux fois la règle de l'affectation à la postcondition $\{I, Tmp\}$: une première fois pour l'affectation $X := \min(X|_{This-1}, X|_{This+1})$, et une seconde fois pour l'affectation $X' := X$. Ces applications consistent en deux substitutions conditionnées par le contexte courant Tmp. Dénotons par $\min_v$ le minimum de $X|_{v-1}$ et $X|_{v+1}$. Après simplification suivant les deux valeurs possibles de Tmp, le prédicat P obtenu est

$$P \equiv \exists k : \forall u : \left\{ \begin{array}{l} \neg\, Tmp|_u \Rightarrow \\ \quad \left\{ \begin{array}{l} X|_u = D|_u \\ \neg\, Tmp|_{u-1} \Rightarrow X|_{u-1} = D|_{u-1} \\ \quad Tmp|_{u-1} \Rightarrow \min_{u-1}+1 = D|_{u-1} \\ \neg\, Tmp|_{u+1} \Rightarrow X|_{u+1} = D|_{u+1} \\ \quad Tmp|_{u+1} \Rightarrow \min_{u+1}+1 = D|_{u+1} \end{array} \right. \\ \\ Tmp|_u \Rightarrow \\ \quad \left\{ \begin{array}{l} \min_u+1 = X|_u \Rightarrow \\ \quad \left\{ \begin{array}{l} \min_u+1 = D|_u \\ \quad \neg\, Tmp|_{u-1} \Rightarrow X|_{u-1} = D|_{u-1} \\ \quad\quad Tmp|_{u-1} \Rightarrow \min_{u-1}+1 = D|_{u-1} \\ \quad \neg\, Tmp|_{u+1} \Rightarrow X|_{u+1} = D|_{u+1} \\ \quad\quad Tmp|_{u+1} \Rightarrow \min_{u+1}+1 = D|_{u+1} \end{array} \right. \\ \\ \min_u+1 \neq X|_u \Rightarrow \\ \quad \left\{ \begin{array}{l} \min_u+1 = k \wedge k \leq D|_u \\ \neg\, Tmp|_{u-1} \Rightarrow X|_{u-1} = k-1 \\ \quad Tmp|_{u-1} \Rightarrow \min_{u-1}+1 = X|_{u-1} \Rightarrow \min_{u-1}+1 = k-1 \\ \neg\, Tmp|_{u+1} \Rightarrow X|_{u+1} = k-1 \\ \quad Tmp|_{u+1} \Rightarrow \min_{u+1}+1 = X|_{u+1} \Rightarrow \min_{u+1}+1 = k-1 \end{array} \right. \end{array} \right. \end{array} \right.$$

Nous ne détaillerons pas ici davantage le calcul de ces deux substitutions.

Il reste finalement à prouver que

$$I \wedge \exists u : X|_u \neq X'|_u \Rightarrow P[(X' \neq X)/Tmp]$$

Preuve. *Fixons une valeur de k qui convienne pour I. Nous allons montrer que $P[(X' \neq X)/Tmp]$ est vrai avec $k+1$ pour valeur de k. Remarquons tout d'abord que la distance vérifie par définition la propriété suivante : $\forall u : D|_u = \min(D|_{u-1}, D|_{u+1}) + 1$. Le prédicat P dans lequel on substitue $X' \neq X$ à Tmp est équivalent à*

$$
P \equiv \exists k : \forall u : \left\{
\begin{array}{l}
X|_u = X'|_u \Rightarrow \\
\quad \text{(i)} \left\{
\begin{array}{l}
X|_u = D|_u \\
X|_{u-1} = X'|_{u-1} \Rightarrow X|_{u-1} = D|_{u-1} \\
X|_{u-1} \neq X'|_{u-1} \Rightarrow \min_{u-1}+1 = D|_{u-1} \\
X|_{u+1} = X'|_{u+1} \Rightarrow X|_{u+1} = D|_{u+1} \\
X|_{u+1} \neq X'|_{u+1} \Rightarrow \min_{u+1}+1 = D|_{u+1}
\end{array}
\right. \\
\\
X|_u \neq X'|_u \Rightarrow \\
\quad \left\{
\begin{array}{l}
\min_u +1 = X|_u \Rightarrow \\
\quad \text{(ii)} \left\{
\begin{array}{l}
\min_u +1 = D|_u \\
X|_{u-1} = X'|_{u-1} \Rightarrow X|_{u-1} = D|_{u-1} \\
X|_{u-1} \neq X'|_{u-1} \Rightarrow \min_{u-1}+1 = D|_{u-1} \\
X|_{u+1} = X'|_{u+1} \Rightarrow X|_{u+1} = D|_{u+1} \\
X|_{u+1} \neq X'|_{u+1} \Rightarrow \min_{u+1}+1 = D|_{u+1}
\end{array}
\right. \\
\\
\min_u +1 \neq X|_u \Rightarrow \\
\quad \text{(iii)} \left\{
\begin{array}{l}
\min_u +1 = k \wedge k \leq D|_u \\
X|_{u-1} = X'|_{u-1} \Rightarrow X|_{u-1} = k-1 \\
X|_{u-1} \neq X'|_{u-1} \Rightarrow \min_{u-1}+1 = X|_{u-1} \Rightarrow \min_{u-1}+1 = k-1 \\
X|_{u+1} = X'|_{u+1} \Rightarrow X|_{u+1} = k-1 \\
X|_{u+1} \neq X'|_{u+1} \Rightarrow \min_{u+1}+1 = X|_{u+1} \Rightarrow \min_{u+1}+1 = k-1
\end{array}
\right.
\end{array}
\right.
\end{array}
\right.
$$

La preuve se fait en détaillant les différents cas possibles.

(i) *Nous avons* $X|_u = X'|_u$. *Par hypothèse de validité de* I, $X|_u = D|_u$, $X|_{u-1} = D|_{u-1}$, *et* $X|_{u+1} = D|_{u+1}$. *Examinons* $X|_{u-1}$. *Si* $X|_{u-1} = X'|_u$ *alors* $X|_{u-1} = D|_{u-1}$, *sinon* $X|_{u-1} = k$, $X|_u = k-1$ *et* $X|_{u-2} \geq k-1$. *Donc* $D|_{u-1} = \min(X|_{u-2}, X|_u) + 1 = k$. *La preuve est identique pour* $X|_{u+1}$.

(ii) *Nous avons* $X|_u \neq X'|_u$ *et* $\min(X|_{u-1}, X|_{u+1}) = k-1$. *Par hypothèse de validité de* I, $X|_u = k$. *Nous avons alors deux cas à considérer. Dans le premier cas,* $X|_{u-1} = k-1$ *et* $X|_{u+1} \geq k-1$. *Le second cas est symétrique donc nous ne traitons que le premier. Comme* $X|_{u-1} \neq k$, *nous déduisons de l'absurde que* $X|_{u-1} = X'|_{u-1}$. *Donc* $X|_{u-1} = D|_{u-1}$ *et* $X|_u = D|_u = \min(X|_{u-1}, X|_{u+1}) + 1$. *Examinons maintenant* $X|_{u+1}$. *Si* $X|_{u+1} = X'|_{u+1}$, *alors* $X|_{u+1} = k-1 = D|_{u+1}$, *sinon* $X|_{u+1} = k$ *et il faut à nouveau considérer deux cas.*

Soit $X|_{u+2} = X'|_{u+2}$. *Alors* $X|_{u+2} = k-1 = D|_{u+2}$ *et* $X|_{u+1} = k = D|_{u+1}$. *Donc* $\min(X|_u, X|_{u+2}) + 1 = k = D|_{u+1}$.

Soit $X|_{u+2} \neq X'|_{u+2}$. *Alors nous avons* $X|_{u+2} = k \leq D|_{u+2}$ *et* $\min(X|_u, X|_{u+2}) = k$. *Comme* $X|_u = D|_u = k$, *nous avons* $\min(D|_u, D|_{u+2}) = k$. *Donc* $D|_{u+1} = \min(D|_u, D|_{u+1}) + 1 = \min(X|_u, X|_{u+1}) + 1$.

(iii) *Nous avons* $X|_u \neq X'|_u$. *Par hypothèse de validité de* I, $X|_u = k$ *et* $\min(X|_{u-1}, X|_{u+1}) \neq k-1$. *Nous déduisons par l'absurde que* $X|_{u-1} \neq X'|_{u-1}$ *et* $X|_{u+1} \neq X'|_{u+1}$, *c'est-à-dire que* $X|_{u-1} = k$ *et* $X|_{u+1} = k$. *Donc* $\min(X|_{u-1}, X|_{u+1}) + 1 = k+1$. *Comme* $\min(D|_{u-1}, D|_{u+1}) \geq k$, *nous déduisons que* $D|_u \geq k+1$.

Examinons $X|_{u-1}$. *Si* $\min(X|_{u-2}, X|_u) + 1 = X|_{u-1}$, *alors* $\min(X|_{u-2}, X|_u) + 1 = k$. *La preuve est identique pour* $X|_{u+1}$.

❑

Bilan

Nous avons vu que le système de preuve que nous avons défini au chapitre 2 permet d'établir la preuve formelle d'une spécification. Les restrictions que nous avions dû apporter nous ont obligé d'introduire une variable auxiliaire pour stocker temporairement le contexte. Il a ensuite fallu introduire une nouvelle règle pour éliminer cette variable auxiliaire.

D'autre part, la méthode utilisée, qui consiste à construire un arbre de preuve, n'est pas satisfaisante. Les preuves obtenues s'avèrent rapidement illisibles. Comme dans le cas séquentiel, nous cherchons à définir une méthode de *preuve par annotations*. Une telle méthode nous permet d'intercaler les assertions dans le texte du programme. Elle a déjà été utilisée informellement pour la preuve de programmes $\mathcal{L}$, par Bougé et al. dans [11], ou par Utard dans sa thèse ([68]). Nous allons ici en donner une définition précise, générale et justifiée.

Auparavant, il nous faut traiter un autre problème : celui de la *complétude* de notre système de preuve. Plus précisément, il s'agit de montrer que toute spécification valide est dérivable dans le système de preuve. Ce sera l'occasion de définir plus précisément la méthode d'introduction et d'élimination de variables auxiliaires. Ce point apparemment théorique est essentiel, non seulement parce qu'il doit prouver que le système de preuve est suffisamment riche, mais aussi parce qu'il nous fournira les clefs pour définir la méthode de preuves par annotations.

chapitre 4

Une première tentative pour établir la complétude[1]

Nous disposons pour l'instant d'un système de preuve qui nous permet de dériver formellement des spécifications. Nous savons que ce système est correct, c'est-à-dire que les spécifications dérivées sont toutes valides. Nous abordons maintenant le problème inverse : celui de la *complétude*. En d'autres mots, pouvons nous dériver *toute* spécification valide?

Cette question a bien sûr un intérêt théorique. Il est intéressant de savoir si le système que nous voulons utiliser nous permettra de traiter tous les cas, même si à première vue, sur quelques exemples, il semble « fonctionner ». Et ceci d'autant plus que nous avons été obligé d'apporter des restrictions aux règles de preuve garantissant que les variables des expressions de contexte ne sont pas modifiées. Mais ce n'est pas son seul intérêt. Nous allons voir qu'essayer de prouver la complétude est l'occasion d'acquérir une compréhension fine du fonctionnement du système de preuve. Dans ce cas précis, elle nous fera prendre conscience des lacunes de cette première approche, que nous essaierons de corriger de plusieurs façons. Enfin, et ce n'est pas le moindre intérêt, cette étude nous donnera une piste pour la recherche d'une méthode de preuves par annotations.

Méthode Partons d'une spécification *valide*

$$\models \{P, C\}\ S\ \{Q, D\}$$

Nous voulons dériver formellement cette spécification, donc obtenir

$$\vdash \{P, C\}\ S\ \{Q, D\}$$

Une méthode usuelle consiste à trouver une « entité » (nous ne connaissons pas encore sa nature), que pour l'instant nous notons ⍰, telle que l'on soit toujours capable de prouver

$$\vdash \{⍰\}\ S\ \{Q, D\}$$

et telle que l'on ait

$$\{P, C\} \Rightarrow ⍰$$

afin que l'on puisse appliquer la règle de conséquence. On remarquera que le fait de privilégier la postcondition plutôt que la précondition vient de ce que l'axiome d'affectation se lit « en arrière ».

1. La matière de ce chapitre est tirée de *On the completeness of a proof system for a simple data-parallel language* (L. Bougé et D. Cachera, [7]).

Un candidat parfait pour cette « entité » est la *plus faible précondition*, c'est-à-dire le plus grand ensemble d'états tel que, partant de l'un de ces états, l'état final obtenu (s'il existe) valide la postcondition. Cette notion est définie plus précisément dans la suite. Ce qu'il est important de savoir pour l'instant, c'est que cet ensemble d'états vérifie « naturellement », à partir du moment où la spécification est valide, l'implication $\{P, C\} \Rightarrow ⍰$, sous la forme

$$[\![\{P, C\}]\!] \subseteq ⍰$$

Nous disposons donc d'un ensemble d'états qui pourrait convenir. Cependant, il nous reste à résoudre deux problèmes :

- ⍰ doit être utilisable dans notre système de preuve. Il faut donc que cet ensemble d'états soit exprimable par une assertion. C'est le problème de la *définissabilité* des plus faibles préconditions.
- Il nous faudra ensuite prouver que

  $$\{⍰\}\ S\ \{Q, D\}$$

  est dérivable dans le système de preuve.

Nous traitons ces deux problèmes dans ce chapitre, en commençant par celui de la définissabilité.

4.1 Plus faibles préconditions et définissabilité

Définitions La *plus faible précondition* d'un programme S par rapport à un ensemble d'états $\mathcal{E}$, notée[2] $wlp(S, \mathcal{E})$, est l'ensemble de tous les états s tels que, si S est exécuté à partir de s et si S termine, alors l'état final résultant est dans $\mathcal{E}$. La plus faible précondition *stricte* d'un programme S par rapport à un ensemble d'états $\mathcal{E}$, notée $wp(S, \mathcal{E})$, est l'ensemble de tous les états s différents de $\perp$ tels que, si S est exécuté à partir de s, *alors* S *termine* et l'état final résultant est dans $\mathcal{E}$. En voici les définitions formelles.

Définition 6 (Plus faibles préconditions) *Soit $\mathcal{E}$ un ensemble d'états, S un programme $\mathcal{L}$. La* plus faible précondition *de S par rapport à $\mathcal{E}$ est définie par*

$$wlp(S, \mathcal{E}) = \{s \mid [\![S]\!](s) \in \mathcal{E} \cup \{\perp\}\}$$

et la plus faible précondition stricte *par*

$$wp(S, \mathcal{E}) = wlp(S, \mathcal{E}) \cap \{s \mid s \neq \perp \Rightarrow [\![S]\!](s) \neq \perp\}$$

Nous allons appliquer cette notion de plus faible précondition à des ensembles d'états particuliers, puisqu'il s'agit d'ensembles d'états définis par des postconditions. Pour alléger les notations, on écrira souvent

$$wp(S, \{Q, D\})$$

au lieu de

$$wp(S, [\![\{Q, D\}]\!])$$

2. La notation vient de *weakest liberal precondition*.

La première propriété importante est que, par définition, les plus faibles préconditions vérifient le *lemme de conséquence* usuel : la précondition d'une spécification valide est incluse dans la plus faible précondition associée à cette spécification.

Lemme 2 (Lemme de conséquence)

$$\models \{P, C\}\ S\ \{Q, D\} \quad ssi \quad [\![\{P, C\}]\!] \subseteq wlp(S, \{Q, D\}).$$

Nous ne donnons pas ici la preuve de ce lemme, qui est immédiate [5].

Restrictions La propriété de *définissabilité* dit que la plus faible précondition relative à un ensemble d'états finaux décrit par une assertion peut elle aussi être décrite par une assertion. Dans notre cas, le problème de définissabilité peut être formalisé par la définition suivante.

Définition 7 (Définissabilité) *Le langage des assertions en deux parties a la propriété de définissabilité si pour tout programme S et toute assertion $\{Q, D\}$, il existe une assertion $\{P, C\}$ telle que*

$$wlp(S, [\![\{Q, D\}]\!]) = [\![\{P, C\}]\!]$$

On acceptera l'abus de notation suivant : $wlp(S, \{Q, D\}) = \{P, C\}$.

Cette propriété est vérifiée dans le cadre de la logique de Hoare classique. Dans ce cadre, la seule difficulté provient du traitement des boucles. Le langage d'assertion doit être suffisamment expressif pour coder dans un prédicat le fait qu'une boucle effectue un certain nombre d'itérations, comme le montrent par exemple Apt et Olderog ([1]). Qu'en est-il dans le cas de notre langage d'assertion ? Deux problèmes vont cohabiter : celui des boucles, semblable au cas de la logique de Hoare classique, et celui, spécifique à notre cas, du traitement de l'affectation data-parallèle, et plus précisément de la partie contexte. Nous allons voir que malheureusement on ne peut obtenir la propriété de définissabilité. En effet, la forme que nous avons donnée aux assertions limite leur pouvoir expressif. Le fait de décrire le contexte par une expression indépendante du prédicat rend celui-ci *fonctionnellement* dépendant de l'environnement, c'est-à-dire des valeurs des variables. Pour un environnement donné, il ne peut y avoir qu'un contexte.

Fait (Expressivité réduite des assertions) *Soit $\{P, C\}$ une assertion. Pour tout environnement σ, il existe au plus un contexte c tel que $(\sigma, c) \models \{P, C\}$: précisément, $c = \sigma(C)$.*

En conséquence de ce fait, les plus faibles préconditions d'un programme $\mathcal{L}$ ne pourront pas toujours être définies par une assertion. Considérons par exemple le programme suivant

$$S \equiv \mathsf{loop}\ \ True\ \ \mathsf{do}\ \ X := X\ \ \mathsf{end},$$

et associons-lui la postcondition $\{true, True\}$. Cette postcondition est satisfaite soit si S termine avec un contexte actif partout, soit si S diverge. La première possibilité est exclue, d'après la sémantique de la boucle. La seconde se produit dès qu'il existe au moins un indice actif : le contexte c doit vérifier $\exists u : C|_u = true$. On a donc

$$wlp(S, \{true, True\}) = \{(\sigma, c) \mid \exists u : c|_u = true\} \cup \{\bot\}$$

Pour le même environnement σ, deux contextes différents (au moins) peuvent produire une divergence. L'ensemble d'états défini par la plus faible précondition ne peut donc pas être décrit par une assertion.

En revanche, si l'on considère la plus faible précondition *stricte*, il faut exclure la divergence, et l'on a

$$wp(S, \{true, True\}) = \{\bot\}$$

Cette fois-ci, l'ensemble d'états peut être décrit par une assertion : $[\![\{false, False\}]\!]$ convient.

Nous pourrions penser que la propriété de définissabilité est valable pour les plus faibles préconditions strictes. Ce n'est malheureusement pas le cas. Considérons le programme

$$S \equiv X := X + 1$$

et la précondition $wp(S, \{Q, D\})$, avec

$$Q \equiv (\forall u : X|_u = 1) \vee (\forall u : X|_u = 2) \qquad \text{et} \qquad D \equiv (X = 2)$$

Soit σ un environnement tel que $\forall u : \sigma(X)|_u = 1$. Soit σ' l'environnement obtenu à partir de σ après exécution de S. Si tous les indices sont actifs, alors l'affectation a lieu partout, et σ' vérifie $\forall u : \sigma'(X)|_u = 2$. Donc D s'évalue en *True* dans σ' et l'état final satisfait $\{Q, D\}$. Si tous les indices sont inactifs, rien ne change par rapport à l'environnement initial : $\forall u : \sigma'(X)|_u = 1$. Cette fois, D s'évalue en *False* dans σ', et l'état final satisfait encore $\{Q, D\}$. On a donc

$$(\sigma, True) \in wp(S, \{Q, D\}) \text{ and } (\sigma, False) \in wp(S, \{Q, D\})$$

S'il existait une assertion $\{P, C\}$ pour décrire $wp(S, \{Q, D\})$, alors on aurait à la fois $\sigma(C) = True$ et $\sigma(C) = False$, ce qui est impossible.

Une première méthode pour traiter ces problèmes de définissabilité consiste à restreindre les programmes et les spécifications que nous considérons. Nous commençons par traiter des cas simples, puis nous tenterons de généraliser les résultats obtenus. Nous nous limitons tout d'abord au cas simple de la logique de Hoare : celui des programmes sans boucles. Ceci nous permet de nous concentrer sur la spécificité du problème data-parallèle. Nous donnons la définition suivante.

Définition 8 (Programmes linéaires) *Un programme S est dit* linéaire *s'il n'est composé que d'affectations, de séquencements et de conditionnements. Les programmes linéaires sont donc les programmes sans boucles.*

Un programme linéaire ne peut naturellement pas diverger. Dans ce cas, les plus faibles préconditions et les plus faibles préconditions strictes sont identiques. Nous ne ferons donc pas la distinction. Les exemples que nous venons de traiter ci-dessus montrent que, hormis le problème des boucles, les difficultés proviennent des interactions entre variables en partie gauche des affectations et variables présentes dans la partie contexte des assertions. Nous introduisons donc la restriction suivante.

Définition 9 (Spécification simple) *Un couple $(S, \{Q, D\})$ est dit* simple *si l'on a*

$$Var(D) \cap Change(S) = \emptyset$$

Une formule de spécification $\{P, C\}\ S\ \{Q, D\}$ est dite simple si le couple $(S, \{Q, D\})$ correspondant est simple.

Nous allons voir maintenant qu'il est possible d'établir des résultats partiels de définissabilité, qui nous permettront de prouver la complétude et de justifier la définition des règles de preuve.

Constructions de base Nous commençons avec les cas simples de l'affectation et de la composition séquentielle.

Proposition 3 (Affectation) *Si* $X \notin Var(D)$, *alors*

$$wp(X := E, \{Q, D\}) = \{Q[(D?E : X)/X], D\}$$

Proposition 4 (Composition séquentielle)

$$wp(S; T, \{Q, D\}) = wp(S, wp(T, \{Q, D\}))$$

La preuve de ces deux propositions est immédiate, d'après la définition des plus faibles préconditions et la sémantique des constructions.

Dans le cas du conditionnement, nous pouvons également donner un résultat simple à condition de supposer que le corps du conditionnement ne modifie pas les expressions de contexte.

Proposition 5 *Supposons que* $(S, \{Q, D \wedge B\})$ *est simple. Si*

$$wp(S, \{Q, D \wedge B\}) = \{P, C\}$$

alors

$$wp(\textsf{where } B \textsf{ do } S \textsf{ end}, \{Q, D\}) = \{P, D\}$$

La preuve de cette proposition utilise un lemme technique : si la plus faible précondition stricte d'un programme et d'une postcondition peut être définie par une assertion, et si l'expression de contexte de la postcondition n'est pas modifiable par le programme, alors cette même expression peut être utilisée pour décrire le contexte dans l'assertion définissant la plus faible précondition. Formellement, ceci se traduit en

Lemme 3 (Extension) *Supposons que* $(S, \{Q, D\})$ *est simple. Si*

$$wp(S, \{Q, D\}) = \{P, C\}$$

alors

$$wp(S, \{Q, D\}) = \{P, D\}$$

Ce lemme est intuitivement clair. Il ne faut cependant pas oublier de remarquer qu'il n'est valable que dans le cas où le programme ne diverge pas à partir de la précondition $\{P, C\}$. Prenons l'exemple du programme

$$S \equiv \textsf{loop } X = 0 \textsf{ do } Y := Y \textsf{ end}$$

Nous avons

$$\models \{\exists u : X|_u = 0, X = 0\}\ S\ \{\forall u : X|_u \neq 0, X \neq 0\}$$

mais

$$\not\models \{\exists u : X|_u = 0, X \neq 0\}\ S\ \{\forall u : X|_u \neq 0, X \neq 0\}.$$

Nous donnons maintenant la preuve de la proposition 5, certes technique, mais assez caractéristique.

Preuve. *Soit* $(\sigma, c) \in wp(\textsf{where } B \textsf{ do } S \textsf{ end}, \{Q, D\})$. *Par définition,*

$$[\![\textsf{where } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = (\sigma', c) \in [\![\{Q, D\}]\!],$$

avec σ' tel que $(\sigma', c \wedge \sigma(B)) = [\![S]\!](\sigma, c \wedge \sigma(B))$. Comme $(Var(D) \cup Var(B)) \cap Change(S) = \emptyset$, nous avons

$$\sigma'(D) = \sigma(D) = c \text{ et } \sigma'(B) = \sigma(B).$$

Donc, $c \wedge \sigma(B) = \sigma'(D \wedge B)$, et $(\sigma', c \wedge \sigma(B)) \in [\![\{Q, D \wedge B\}]\!]$. Par hypothèse, nous avons $(\sigma, c \wedge \sigma(B)) \in [\![\{P, C\}]\!]$, et $\sigma \models P$. Donc nous avons bien $(\sigma, c) \in [\![\{P, D\}]\!]$.

Réciproquement, soit $(\sigma, c) \in [\![\{P, D\}]\!]$. Remarquons d'abord que, par le lemme 3,

$$wp(S, \{Q, D \wedge B\}) = \{P, C\} = \{P, D \wedge B\}.$$

Donc $[\![S]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c') \in [\![\{Q, D \wedge B\}]\!]$, et $\sigma' \models Q$. D'où $[\![\mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}]\!](\sigma, c) = (\sigma', c)$. Comme $Var(D) \cap Change(S) = \emptyset$, on a $\sigma'(D) = \sigma(D) = c$, et $(\sigma', c) \in [\![\{Q, D\}]\!]$. □

Arrivés à ce point, nous disposons de quelques résultats partiels de définissabilité. Ces résultats ne sont bien sûr pas suffisants pour prouver la complétude, mais ils nous éclairent sur la façon dont « fonctionne » le langage. Nous allons maintenant voir comment il est possible de lever les restrictions que nous avons imposées pour simplifier notre première approche.

Introduction de variables auxiliaires Nous avons travaillé pour l'instant sur des structures de conditionnement where B do S end telles que $Var(B) \cap Change(S) = \emptyset$. Levons cette restriction et supposons $Var(B) \cap Change(S) \neq \emptyset$. Supposons de plus qu'il existe une « nouvelle » variable Tmp qui n'apparaisse dans aucun programme ou expression utilisés jusqu'ici. Transformons alors le programme where B do S end en

$$Tmp := B; \text{ where } Tmp \text{ do } S \text{ end}$$

Supposons qu'il existe une assertion $\{P, C\}$ telle que $wp(S, \{Q, D \wedge Tmp\}) = \{P, C\}$. Les résultats précédents nous permettent d'écrire

$$\begin{aligned} &wp(Tmp := B; \text{ where } Tmp \text{ do } S \text{ end}, \{Q, D\}) \\ &= \{P[(D?B : Tmp)/Tmp], D\} \end{aligned}$$

La « nouvelle » variable Tmp joue ici le rôle d'une variable temporaire servant à stocker la valeur de l'expression de contexte. Nous appelons une telle variable une variable *auxiliaire*. Il est nécessaire de pouvoir éliminer de telles variables lorsqu'elles ne sont plus nécessaires, c'est-à-dire à l'extérieur de la structure de conditionnement. Or nous remarquons sur le calcul que nous venons d'effectuer que la variable Tmp est toujours présente dans la précondition. Ceci vient du fait que l'affectation de Tmp n'a lieu que pour les indices actifs. Pour les indices inactifs, l'« ancienne » valeur de Tmp est conservée, alors qu'elle ne fait pas réellement sens. En fait, il est possible d'obtenir un meilleur résultat en encapsulant l'utilisation de la variable auxiliaire directement dans le calcul de la plus faible précondition [3]. Ceci est exprimé par la proposition suivante.

Proposition 6 *Soit $(S, \{Q, D\})$ un couple simple, et soit Tmp une variable telle que $Tmp \notin Var(Q) \cup Var(S) \cup Var(D)$. Si*

$$wp(S, \{Q, D \wedge Tmp\}) = \{P, C\}$$

alors

$$wp(\text{where } B \text{ do } S \text{ end}, \{Q, D\}) = \{P[B/Tmp], C\}$$

3. Nous verrons plus loin (5.2) qu'une autre méthode consiste à améliorer ce mécanisme de transformation syntaxique des programmes.

Preuve. *Afin de ne pas trop nous perdre dans des détails techniques, nous donnons la preuve complète en annexe. Signalons seulement ici que la preuve utilise le fait que D n'est pas modifiable car $(S, \{Q, D\})$ est simple, et que la valeur de Tmp n'est pas modifiée car Tmp n'est pas une variable du programme.* ❑

Définissabilité pour les programmes linéaires Nous pouvons désormais combiner ces premiers résultats pour énoncer le théorème suivant.

Théorème 1 (Définissabilité restreinte) *Soit S un programme linéaire, et $\{Q, D\}$ une assertion telle que $(S, \{Q, D\})$ est simple. Il existe un prédicat P tel que*

$$wp(S, \{Q, D\}) = \{P, D\}.$$

Preuve. *La preuve se fait simplement par induction sur la structure du programme, et combine les résultats des propositions précédentes. Les cas de l'affectation et du séquencement sont immédiats. Le cas du séquencement est traité de la façon suivante : si $S \equiv$* where *B* do *T* end*, on choisit une nouvelle*[4] *variable Tmp telle que $Tmp \notin Var(Q) \cup Var(S) \cup Var(D)$. Par hypothèse d'induction, il existe une assertion $\{P, C\}$ telle que $wp(T, \{Q, D \wedge Tmp\}) = \{P, C\}$. La proposition 6 nous permet d'affirmer que $wp(S, \{Q, D\}) = \{P[B/Tmp], D\}$.* ❑

Ce théorème restreint est dans le cadre que nous avons choisi pour l'instant, le meilleur résultat que nous puissions espérer. L'exemple en page 46 montre en effet que le résultat n'est pas vrai si le couple $(S, \{Q, D\})$ n'est pas simple.

4.2 Complétude partielle

Nous allons utiliser les propriétés de définissabilité partielle pour établir un premier résultat de complétude. Ce résultat n'est pas satisfaisant en ce qui concerne la théorie, puisqu'il ne pourra être appliqué aux programmes contenant des boucles. Il est en revanche intéressant du point de vue de la pratique, puisqu'il nous permettra de définir une méthodologie de preuve. Nous avons vu dans la section précédente comment traiter (partiellement) le problème de la définissabilité. L'ensemble d'états que nous avions noté ⍰ au début de ce chapitre peut donc dans certains cas être défini par une assertion. Reste à traiter l'autre partie du problème : montrer que la formule $\{⍰\}\ S\ \{Q, D\}$ est prouvable. La preuve de la complétude est construite de façon incrémentale, en partant de cas simples, et en levant progressivement les restrictions. L'intérêt de cette démarche réside dans le fait qu'elle va nous permettre de suivre le même cheminement que lors du traitement du problème de la définissabilité, puis d'aller plus loin, puisque nous prouverons la complétude pour les spécifications non simples.

Un premier résultat Les restrictions que nous avons dû imposer en ce qui concerne les résultats de définissabilité nous suggèrent de traiter d'abord les programmes *linéaires*, et les spécifications *simples*. La propriété de définissabilité concernant les structures de conditionnement et faisant intervenir des variables auxiliaires n'est pas directement applicable ici. En effet, la règle de preuve que nous avons énoncée pour les structures de conditionnement ne fait pas intervenir de variable auxiliaire. Nous introduisons donc une restriction syntaxique supplémentaire sur les programmes.

4. Remarquons qu'il est toujours possible de trouver une telle variable, car les expressions du programme et les prédicats du langage d'assertion n'en utilisent qu'un nombre fini.

Nous verrons ensuite comment traiter le problème des variables auxiliaires. La condition que nous imposons sur la forme des programmes est explicitée par la définition suivante.

Définition 10 (Programme régulier) *Un programme S de $\mathcal{L}$ est dit* régulier *si pour tout sous-programme de S de la forme* where B do T end, *on a $Var(B) \cap Change(T) = \emptyset$.*

Le résultat de définissabilité de la section précédente peut alors s'exprimer de la façon suivante : dans le cas de programmes réguliers et linéaires, et de spécifications simples, la définissabilité est établie.

Ce résultat nous permet d'énoncer le théorème suivant.

Théorème 2 (Complétude, cas simple, linéaire, régulier) *Soit $\{P, C\}\ S\ \{Q, D\}$ une spécification simple, où S est un programme linéaire et régulier. Si*

$$\models \{P, C\}\ S\ \{Q, D\}$$

alors

$$\vdash \{P, C\}\ S\ \{Q, D\}.$$

Preuve. *La preuve est classique et suit le schéma de preuve donné par Apt et Olderog ([1]). Nous en donnons ici l'idée, une preuve plus complète étant fournie en annexe. Nous utilisons le calcul des plus faibles préconditions. Pour tout programme S linéaire et régulier, et pour tout couple simple $(S, \{Q, D\})$, il existe d'après le théorème de définissabilité une assertion $\{P', C'\}$ telle que $[\![\{P', C'\}]\!] = wp(S, [\![\{Q, D\}]\!])$. Il faut alors montrer par induction sur la structure de S que $\vdash \{wp(S, \{Q, D\})\}\ S\ \{\{Q, D\}\}$. La preuve est ensuite achevée par l'application de la règle de conséquence et du lemme de conséquence.* ❑

Introduction d'une nouvelle règle Dans le cas des programmes non réguliers, nous avons vu dans la section précédente que nous ne pouvions pas directement trouver une assertion pour exprimer la précondition la plus faible. Nous avons vu également qu'il est possible de s'inspirer d'un mécanisme de transformation syntaxique pour introduire des nouvelles variables qui serviront à stocker la valeur de l'expression de contexte. De la même façon que nous avons encapsulé ces variables dans le calcul des plus faibles préconditions, nous allons les introduire dans le système de preuve. À la notion syntaxique de variable auxiliaire, nous faisons correspondre la notion sémantique de variable *cachée*. Une variable cachée est une variable manipulée dans le système de preuve, mais qui n'apparaît pas dans le programme ou sa spécification. De la même façon que Gries et Owicki ont introduit une règle d'élimination des variables auxiliaires pour la preuve de programmes à parallélisme disjoint [55], nous avons besoin ici d'une règle d'*élimination des variables cachées*. Nous avions introduit cette règle afin de pouvoir traiter l'exemple du chapitre précédent. Nous la rappelons ici.

Règle 6 (Rappel : élimination des variables cachées)

$$\frac{\{P, C\}\ S\ \{Q, D\}, \quad Tmp \notin Var(S) \cup Var(Q) \cup Var(D)}{\{P[E/Tmp], C[E/Tmp]\}\ S\ \{Q, D\}},$$

où E est une expression vectorielle quelconque.

On notera $\vdash^* \{P, C\}\ S\ \{Q, D\}$ le fait qu'une formule de spécification est dérivable dans le système de preuve étendu avec cette nouvelle règle.

Il nous faut vérifier que l'introduction de cette nouvelle règle conserve la correction du système de preuve. Nous avons en effet le théorème suivant.

Théorème 3 (Correction de $\vdash^*$) *Le système de preuve $\vdash^*$ est correct : si*

$$\vdash^* \{P, C\}\ S\ \{Q, D\}$$

alors

$$\models \{P, C\}\ S\ \{Q, D\}.$$

Preuve. *Il suffit de prouver la correction de la nouvelle règle d'élimination. C'est simplement la traduction en termes techniques du fait que si Tmp n'apparaît ni dans le programme ni dans la postcondition, alors il peut être initialisé avec n'importe quelle valeur. Nous ne donnons pas les détails de la preuve.* □

Généralisation du résultat Nous allons utiliser la règle d'élimination pour généraliser notre premier résultat de complétude. Commençons par traiter les programmes non réguliers.

Théorème 4 (Complétude, cas simple, linéaire) *Soit $\{P, C\}\ S\ \{Q, D\}$ une spécification simple, où S est un programme linéaire. Si*

$$\models \{P, C\}\ S\ \{Q, D\}$$

alors

$$\vdash^* \{P, C\}\ S\ \{Q, D\}.$$

Preuve. *La preuve est semblable à celle du théorème 2 pour les programmes réguliers, et se construit par induction sur la structure de S. Le seul nouveau cas que nous devons considérer est celui où $S \equiv$ where B do T end, avec $Var(B) \cap Change(T) \neq \emptyset$. Choisissons une « nouvelle » variable Tmp telle que $Tmp \notin Var(Q) \cup Var(S) \cup Var(D)$. Une telle variable existe car les expressions du programme et celles du langage d'assertion sont des termes finis. Par le théorème 1, nous savons qu'il existe une assertion*

$$\{P, C\} = wp(T, \{Q, D \wedge Tmp\})$$

Nous avons $Var(D \wedge Tmp) \cap Change(S) = \emptyset$ d'après le choix de Tmp. Donc, par le lemme d'extension, $wp(T, \{Q, D \wedge Tmp\}) = \{P, D \wedge Tmp\}$. Par hypothèse d'induction, nous avons

$$\vdash^* \{P, D \wedge Tmp\}\ T\ \{Q, D \wedge Tmp\}$$

Nous avons également $\{P \wedge B = Tmp, D \wedge B\} \Rightarrow \{P, D \wedge Tmp\}$. Nous pouvons donc appliquer la règle de conséquence, qui donne

$$\vdash^* \{P \wedge B = Tmp, D \wedge B\}\ T\ \{Q, D \wedge Tmp\}$$

Nous appliquons alors la règle du where *pour obtenir*

$$\vdash^* \{P \wedge B = Tmp, D\}\ \textsf{where}\ B\ \textsf{do}\ T\ \{Q, D\}$$

Grâce à la règle de conséquence, ceci se réécrit en

$$\vdash^* \{P[B/Tmp] \wedge B = Tmp, D\}\ \textsf{where}\ B\ \textsf{do}\ T\ \{Q, D\}$$

Finalement, l'application de la règle d'élimination avec $E \equiv B$ donne

$$\vdash^* \{P[B/Tmp], D\} \text{ where } B \text{ do } T \ \{Q, D\}$$

D'après la proposition 6, $wp(S, \{Q, D\}) = \{P[B/Tmp], D\}$. Donc

$$\vdash^* wp(S, \{Q, D\}) \ S \ \{Q, D\}.$$

Comme pour le premier théorème de complétude, nous concluons la preuve en appliquant le lemme 2 et la règle de conséquence. ❑

Nous avons traité le problème des programmes non réguliers grâce à l'introduction de la règle d'élimination. Nous allons voir que cette règle nous permet également de lever la restriction que nous avons imposée aux spécifications. Nous sommes en effet en mesure de prouver le théorème suivant.

Théorème 5 (Complétude, programmes linéaires) *Soit S un programme linéaire. Si*

$$\models \{P, C\} \ S \ \{Q, D\}$$

alors

$$\vdash^* \{P, C\} \ S \ \{Q, D\}.$$

Preuve. *Supposons $\models \{P, C\} \ S \ \{Q, D\}$. Comme les expressions du langage d'assertion et du programme sont des termes finis, il existe une « nouvelle » variable cachée Tmp telle que $Tmp \notin Var(S) \cup Var(Q) \cup Var(D)$. Montrons que*

$$\models \{P \wedge Tmp = C, C\} \ S \ \{Q \wedge Tmp = D, Tmp\}$$

Soit (σ, c) dans $[\![\{P \wedge Tmp = C, C\}]\!]$. Nous avons en particulier $(\sigma, c) \models \{P, C\}$. D'après notre hypothèse, nous avons donc $[\![S]\!](\sigma, c) = (\sigma', c) \models \{Q, D\}$.

De plus, nous avons $\sigma(Tmp) = \sigma(C) = c$. Comme $Tmp \notin Var(S)$, nous avons $\sigma'(Tmp) = \sigma(Tmp) = c$, et $(\sigma', c) \models \{Q, D\}$ donne $\sigma'(D) = c$. Nous en concluons que $(\sigma', c) \models \{Q \wedge Tmp = D, Tmp\}$.

Comme $Tmp \notin Var(S)$, nous sommes dans le cas d'une spécification simple. Nous pouvons donc appliquer le théorème 4 de complétude, qui donne

$$\vdash^* \{P \wedge Tmp = C, C\} \ S \ \{Q \wedge Tmp = D, Tmp\}$$

Comme $\{Q \wedge Tmp = D, Tmp\} \Rightarrow \{Q, D\}$, nous pouvons appliquer la règle de conséquence. Nous obtenons

$$\vdash^* \{P \wedge Tmp = C, C\} \ S \ \{Q, D\}$$

En appliquant alors la règle d'élimination avec $E \equiv C$, nous obtenons

$$\vdash^* \{P \wedge C = C, C\} \ S \ \{Q, D\}$$

Finalement, comme $\{P, C\} \Rightarrow \{P \wedge C = C, C\}$, nous en déduisons par ue nouvelle application de la règle de conséquence que

$$\vdash^* \{P, C\} \ S \ \{Q, D\}$$

❑

4.3 Bilan

Nous avons exposé une première approche visant à prouver la complétude de notre système de preuve. Cette approche utilise un calcul classique de plus faibles préconditions. Les propriétés de définissabilité partielle de ce calcul nous ont permis d'établir la complétude dans le cas des programmes sans boucles, à condition d'ajouter une règle d'élimination des variables cachées.

Il n'est malheureusement pas possible d'étendre ce résultat aux programmes avec boucles : nous avons vu au début de ce chapitre que dans le cas où le programme diverge il n'est pas possible d'exprimer les plus faibles préconditions (libérales) par une assertion. Ceci vient du fait que le contexte dépend fonctionnellement de l'environnement, c'est-à-dire des valeurs des variables. Nous verrons au chapitre 6 comment la définition d'un langage d'assertion légèrement différent permet de pallier à ces insuffisances théoriques.

Cependant, cette première étude permet de dégager une idée de méthodologie de preuve. En effet, la règle de substitution des variables cachées suggère d'utiliser des variables auxiliaires pour mémoriser le contexte. Nous développons cette idée dans le chapitre suivant, où est définie une méthode de preuve par annotations.

chapitre 5

Preuves par annotations[1]

Nous avons jusqu'à maintenant travaillé directement avec les règles d'inférence du système de preuve. L'exemple que nous avons traité au chapitre 3 montre que ce type de dérivation est particulièrement ardu à suivre. Pour appliquer de façon pratique les résultats que nous avons obtenus, il nous reste à définir une méthode de preuve plus lisible.

L'idée consiste à utiliser le fait que les règles de preuve suivent la structure syntaxique du programme. La preuve peut donc être donnée en intercalant les assertions *dans le texte du programme* aux endroits appropriés. Ce type de preuve est plus simple à produire et à analyser que la construction d'un arbre dérivé des règles d'inférences, qui nécessite un découpage de la preuve en petites étapes.

Cette méthode est largement utilisée pour les langages scalaires. Introduite par Floyd pour un langage de type Algol [31] sous le nom de *preuve par annotations*[2], elle a été étendue par Gries et Owicki [55] à un langage à parallélisme de contrôle et à mémoire partagée, puis par Apt et Olderog [1] pour les programmes à parallélisme de contrôle et à mémoire distribuée. Nous montrons ici que cette approche peut être adaptée au cas data-parallèle. Nous allons donc définir des annotations data-parallèles et montrer la validité d'ue méthode de preuve par annotations. Le principal avantage de cette méthode est de fournir des preuves aussi « simples » que pour les programmes séquentiels scalaires.

Nous présentons d'abord une méthode de preuve par annotations inspirée de l'étude du langage que nous avons menée au chapitre précédent. Nous présentons ensuite une approche voisine, reposant sur un mécanisme de transformation syntaxique des programmes.

Les résultats que nous avons exposés au chapitre précédent ne s'appliquent qu'aux programmes sans boucles. Nous exposerons d'abord la méthode d'annotations pour les programmes linéaires. Une extension au cas plus général des programmes avec boucles est traitée à la fin de ce chapitre.

5.1 Une méthode de preuve en deux phases

Revenons un instant aux résultats du chapitre précédent. Le calcul des plus faibles préconditions montre que, sous certaines conditions, il est possible de déduire la précondition de la postcondition. Ceci suggère une méthode de preuve « en arrière », c'est-à-dire en partant de la postcondition de la spécification et en générant des assertions tout en remontant dans le texte du programme.

1. La matière de ce chapitre est tirée de *Proving Data-Parallel Programs Correct: The Proof Outlines Approach* (L. Bougé et D. Cachera, [8]).

2. Proof Outline dans sa dénomination originale.

L'étude précédente montre aussi qu'il est parfois nécessaire d'introduire des variables auxiliaires pour stocker les valeurs des expressions booléennes de contexte. Ceci nous suggère une méthode de preuve en deux phases : d'abord un étiquetage syntaxique du programme pour traiter la partie contexte des assertions, puis une preuve « en arrière » semblable à la preuve d'un programme séquentiel scalaire.

Première phase : étiquetage syntaxique Cette première étape consiste à étiqueter chaque sous-programme avec un entier qui correspond au nombre de structures de conditionnement englobantes. En un point du programme, le label dénote donc la profondeur d'imbrication du conditionnement. Cet étiquetage suit la syntaxe du programme : le label est initialisé à 0 pour le programme complet, et est incrémenté de 1 à chaque fois que l'on entre dans une nouvelle structure de conditionnement, lors d'un parcours en profondeur de l'arbre syntaxique du programme. Considérons par exemple le programme suivant

```
where X > 0 do
    X := X + 1;
    where X > 2 do
        X := X + 1;
    end
end
```

Nous voulons obtenir l'étiquetage suivant

```
(0) where X > 0 do
    (1)X := X + 1;
    (1) where X > 2 do
        (2)X := X + 1
    end
end
```

Pour mémoriser les expressions de contexte, nous allons distinguer un ensemble particulier de variables qui n'apparaissent pas dans les programmes. Ce sont les variables auxiliaires, qui ont été introduites informellement dans le chapitre précédent, et dont nous donnons la définition suivante.

Définition 11 (Variables auxiliaires) *Les variables* $\{Tmp_i \mid i \in \mathbb{N}\}$ *sont telles que pour tout programme* S *et pour tout entier* i, $Tmp_i \notin Var(S)$. *Cet ensemble est l'ensemble des* variables auxiliaires.

La structure de conditionnement peut être vue comme un mécanisme de pile. Entrer dans un conditionnement correspond à empiler une valeur sur la pile de contexte, alors que sortir du conditionnement consiste à dépiler. L'étiquette que nous avons donnée à un sous-programme correspond donc à la hauteur de la pile de contexte. Chaque variable auxiliaire est utilisée pour stocker une cellule de la pile de contexte : la variable Tmp_i stocke la valeur initiale de l'expression de contexte à la profondeur i. Grâce à ce stockage, les variables apparaissant dans les expressions peuvent être modifiées. Nous pouvons donc lever les restrictions sur les expressions de contexte des structures de conditionnement.

En un certain point du programme, le contexte courant correspond à la conjonction de toutes les valeurs présentes dans la pile, donc à la valeur courante de $\bigwedge_{k=0}^{i} Tmp_i$. Pour traduire ce fait de façon visuelle dans la preuve, nous ajoutons des annotations de la forme $[Tmp_i \equiv B]$ à chaque

where. L'exemple précédent devient alors

$$
\begin{array}{l}
(0)\ \mathsf{where}\ \ X > 0\ \ \mathsf{do}\ \ [Tmp_1 \equiv X > 0] \\
\quad (1)X := X + 1; \\
\quad (1)\ \mathsf{where}\ \ X > 2\ \ \mathsf{do}\ \ [Tmp_2 \equiv X > 2] \\
\quad\quad (2)X := X + 1 \\
\quad \mathsf{end} \\
\mathsf{end}
\end{array}
$$

Dans un souci de complétude, nous donnons maintenant une définition formelle de l'étiquetage des programmes. Cet étiquetage est défini par induction sur la structure du programme, et exprimé par les règles ci-dessous.

$$
\begin{array}{rcl}
\varphi(\ X := E\ , i) & = & (i)\ \ X := E \\
\varphi(\ S\ ;\ T\ , i) & = & \varphi(S, i)\ ;\ \varphi(T, i) \\
\varphi(\ \mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}\ , i) & = & (i)\ \ \mathsf{where}\ B\ \mathsf{do}\ [Tmp_{i+1} \equiv B] \\
 & & \quad \varphi(S, i+1) \\
 & & \quad \mathsf{end}
\end{array}
$$

Pour un programme S, l'étiquetage de S est obtenu en prenant $\varphi(S, 0)$.

Deuxième phase : annotations Une preuve par annotations est un moyen visuel et pratique de présenter une preuve, avec des assertions intercalées dans le texte du programme. La structure de la preuve suit donc celle du programme. Comme nous utilisons des programmes étiquetés et des variables auxiliaires pour mémoriser le contexte, nous connaissons en tout point du programme une expression qui dénote le contexte courant. Nous pouvons donc éliminer les expressions de contexte des assertions et procéder exactement comme dans le cas scalaire, avec des substitutions « en arrière ». Deux petites différences sont à considérer par rapport au cas scalaire : d'une part, les substitutions opérées pour traiter les affectations sont conditionnées par une conjonction de Tmp_k correspondant au contexte, et d'autre part le mécanisme de conditionnement introduit un autre type de substitution qui n'a pas lieu d'être dans le cas scalaire.

Nous donnons en figure 5.1 les règles d'insertion des annotations. La contiguïté de deux assertions traduit l'emploi de la règle de conséquence. Si S est un sous-programme étiqueté, nous notons S^* une annotation de S obtenue en insérant des assertions dans S, et par $Lab(S)$ l'étiquette associée à S. Remarquons que, comme l'étiquetage commence à 0 pour le programme dans son ensemble, le contexte initial dans lequel le programme est exécuté est dénoté par Tmp_0.

Expliquons intuitivement la nécessité d'imposer des restrictions de la forme $\forall j > i, Tmp_j \notin Var(Q)$. Dans la règle de conditionnement, nous substituons Tmp_{i+1} par B. Nous avons donc besoin, pour respecter les conditions de la règle de substitution du système de preuve, d'imposer la restriction $Tmp_{i+1} \notin Var(Q)$. Mais, comme la postcondition (Q) est la même pour S et pour $\mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}$, il faut que cette condition soit satisfaite pour toutes les profondeurs de conditionnement supérieures à $Lab(S)$.

$$\frac{\forall j > i, Tmp_j \notin Var(Q)}{\{Q[\bigwedge_{k=0}^{i} Tmp_k ?E\colon X/X]\}\ \text{(i)}\ X := E\ \{Q\}}$$

$$\frac{\{P\}\ S^*\ \{R\} \quad \{R\}\ T^*\ \{Q\} \quad \forall j > Lab(S), Tmp_j \notin Var(R) \cup Var(Q)}{\{P\}\ S^*; \{R\}\ T^*\ \{Q\}}$$

$$\frac{P \Rightarrow P' \quad \{P'\}\ S^*\ \{Q'\} \quad Q' \Rightarrow Q \quad \forall j > Lab(S), Tmp_j \notin Var(Q) \cup Var(Q')}{\{P\}\{P'\}\ S^*\ \{Q'\}\{Q\}}$$

$$\frac{\begin{array}{c}\{P\}\ S^*\ \{Q\} \\ Lab(S) = i+1 \\ \forall j > i, Tmp_j \notin Var(Q)\end{array}}{\begin{array}{l}\{P[B/Tmp_{i+1}]\}\ \text{(i) where}\ B\ \text{do}\ [Tmp_{i+1} \equiv B] \\ \qquad\qquad\qquad\qquad\quad \{P\} \\ \qquad\qquad\qquad\qquad\quad S^* \\ \qquad\qquad\qquad\qquad\quad \{Q\} \\ \qquad\qquad\qquad\qquad\quad \text{end}\ \{Q\}\end{array}}$$

$$\frac{\{P\}\ S^*\ \{Q\}}{\{P\}\ S^{**}\ \{Q\}}$$

où S^{**} est obtenu à partir de S^* par suppression d'une assertion quelconque.

FIG. 5.1 – *Règles d'annotations*

Exemple Revenons à l'exemple du début de ce chapitre. Nous voulons prouver la spécification suivante.

$\{\forall u : X|_u = 2, True\}$
where $X > 0$ do
$X := X + 1;$
where $X > 2$ do
$X := X + 1$
end
end
$\{\forall u X|_u = 4, True\}$

La preuve est simplement faite en établissant l'annotation suivante.

$\{\forall u : (Tmp_0 \wedge X > 0 \wedge (Tmp_0 \wedge X > 0 ? X + 1 : X) > 2 ?$
$(Tmp_0 \wedge X > 0 ? X + 1 : X) + 1 : (Tmp_0 \wedge X > 0 ? X + 1 : X))|_u = 4\}$
(0) where X>0 do $[Tmp_1 \equiv X > 0]$
$\{\forall u : (Tmp_0 \wedge Tmp_1 \wedge (Tmp_0 \wedge Tmp_1 ? X + 1 : X) > 2 ?$
$(Tmp_0 \wedge Tmp_1 ? X + 1 : X) + 1 : (Tmp_0 \wedge Tmp_1 ? X + 1 : X))|_u = 4\}$
(1) X:=X+1;
$\{\forall u : (Tmp_0 \wedge Tmp_1 \wedge X > 2 ? X + 1 : X)|_u = 4\}$
(1) where X>2 do $[Tmp_2 \equiv X > 2]$
$\{\forall u : (Tmp_0 \wedge Tmp_1 \wedge Tmp_2 ? X + 1 : X)|_u = 4\}$
(2) X:=X+1
$\{\forall u : X|_u = 4\}$
end
$\{\forall u : X|_u = 4\}$
end
$\{\forall u : X|_u = 4\}$

Si nous notons P la première assertion de cette preuve par annotations, nous devons seulement prouver que

$$\forall u : X|_u = 2 \wedge Tmp_0 = True \Rightarrow P.$$

En d'autres mots, nous prouvons que

$$\forall u : X|_u = 2 \Rightarrow P[True/Tmp_0]$$

L'assertion $\{P[True/Tmp_0]\}$ est équivalente à

$$\{(X > 0 \wedge (X > 0 ? X + 1 : X) > 2 ? (X > 0 ? X + 1 : X) + 1 : (X > 0 ? X + 1 : X))|_u = 4\}$$

Considérons un indice u tel que $X|_u = 2$. Alors l'expression booléenne $(X > 0)|_u$ vaut *true*. Comme $X + 1|_u > 2$, $((X > 0?X + 1 : X) > 2)|_u$ vaut également *true*.

L'expression conditionnelle

$$(X > 0 \wedge (X > 0?X + 1 : X) > 2?(X > 0?X + 1 : X) + 1 : (X > 0?X + 1 : X))|_u$$

se simplifie alors en $(X > 0?X + 1 : X) + 1|_u$, qui à son tour se simplifie en $X + 1 + 1|_u$.

L'assertion $\{P[True/\, Tmp_0]\}$ se simplifie donc en $X + 1 + 1|_u = 4$, qui est vraie.

Validité de la méthode La méthode de preuve par annotations n'est valide que dans la mesure où nous montrons qu'elle respecte la sémantique dénotationnelle. Nous allons en fait montrer que d'une certaine façon elle est équivalente à une dérivation dans le système de preuve initial ($\vdash^*$). Cette équivalence nous garantira que toute preuve par annotations définit une spécification valide, et que réciproquement, pour toute formule prouvable dans le système $\vdash^*$ il existe une preuve par annotations. Nous prouvons donc la propriété d'équivalence suivante.

Théorème 6 *Soit* $\{P\}$ (0) S $\{Q\}$ *une formule où* S *est un programme linéaire et telle que pour tout* $j > 0$, $Tmp_j \notin Var(Q)$. *On a*

$$\{P\}\ S^*\ \{Q\} \textit{ est une annotation pour } S$$

$$\Updownarrow$$

$$\vdash^* \{P, Tmp_0\}\ S\ \{Q, Tmp_0\}$$

Nous prouvons en fait la propriété suivante, plus générale.

Proposition 7 *Soit* S *un sous-programme étiqueté par* i, *et* P *et* Q *deux assertions telles que* $\forall j > 0$, $Tmp_j \notin Var(Q)$. *Alors*

$$\{P\}\ S^*\ \{Q\}$$

est une annotation pour S *si et seulement si*

$$\vdash^* \{P, \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}$$

La preuve du sens direct est simple : elle consiste en une induction sur la construction des annotations. La réciproque est plus difficile et repose sur le calcul des plus faibles préconditions. Les deux preuves sont données en annexe A.

5.2 Un mécanisme de transformation syntaxique

La méthode de preuve par annotations que nous avons présentée utilise les variables auxiliaires de façon implicite. Nous ajoutons en effet à l'entrée de chaque **where** une annotation du type $[Tmp_i \equiv B]$ qui indique à quelle expression de conditionnement la variable auxiliaire réfère. Dans un souci de lisibilité, il peut être souhaitable que ces variables auxiliaires apparaissent directement dans le texte du programme. Nous proposons ici un mécanisme de transformation syntaxique des programmes, équivalent à l'approche précédente.

Afin d'obtenir un résultat simple d'équivalence, nous introduisons un nouveau type d'affectation : l'*affectation inconditionnelle*. Cette affectation se comporte de la même façon que l'affectation

data-parallèle « classique », sauf en ce qui concerne le contexte : à chaque indice, qu'il soit actif ou non, la composante de la variable est mise à jour avec la valeur de l'expression.

$$[\![X \equiv E]\!](\sigma, c) = (\sigma', c), \quad \text{avec } \sigma' = \sigma[X \leftarrow E].$$

Une telle instruction correspond par exemple à un everywhere{X:=E} en HYPERC, ou à un all X:=E en MPL. Avant d'entrer dans une structure de conditionnement, nous conservons la valeur de l'expression de contexte dans la variable Tmp_i correspondant à la profondeur de conditionnement. Plus formellement, la transformation syntaxique est définie par la fonction ψ suivante.

$$\begin{aligned}
\psi(X := E, i) &= X := E \\
\psi(S; T, i) &= \psi(S, i); \psi(T, i) \\
\psi(\textsf{where } B \textsf{ do } S \textsf{ end}, i) &= \begin{cases} Tmp_i \equiv B; \\ \textsf{where } Tmp_i \textsf{ do } \psi(S, i+1) \textsf{ end} \end{cases}
\end{aligned}$$

Cette transformation vérifie le fait suivant.

Fait *Soit S un programme et i un entier. Alors $\psi(S, i)$ est un programme régulier.*

Rappelons qu'un programme régulier est un programme où les expressions de conditionnement ne peuvent pas être modifiées par le corps du conditionnement (cf. définition 10). Le transformé syntaxique de S sur lequel nous allons travailler est obtenu en prenant $\psi(S, 0)$. Nous le noterons $\overline{S}$. Nous dirons que $\overline{S}$ est une *régularisation* ou un *régularisé* de S.

Notre but est de montrer que les deux approches, l'une qui encapsule la substitution des variables auxiliaires dans la règle du where, et l'autre qui explicite cette substitution dans la syntaxe, sont équivalentes.

La première propriété que nous devons vérifier est l'équivalence sémantique d'un programme avec son transformé syntaxique. Considérons $\{P, C\}$ et $\{Q, D\}$ deux assertions telles que pour toute variable auxiliaire Tmp_i, nous ayons $Tmp_i \notin Var(P) \cup Var(C) \cup Var(Q) \cup Var(D)$. Nous avons alors

$$\models \{P, C\}\ S\ \{Q, D\} \iff \models \{P, C\}\ \overline{S}\ \{Q, D\}.$$

Preuve. *Il faut en fait prouver un résultat plus général:*

$$\textit{pour tout } i \geq 0,\ \models \{P, C\}\ S\ \{Q, D\} \iff \models \{P, C\}\ \varphi(S, i)\ \{Q, D\}.$$

La preuve se fait simplement par induction sur la structure de S, et utilise le fait que les variables auxiliaires ne sont pas modifiées par les instructions. Nous n'en donnerons pas les détails ici. ❑

Nous pouvons maintenant définir des règles pour annoter les programmes régularisés $\overline{S}$ (cf. figure 5.2). Les règles de l'affectation sont inchangées. Celle du conditionnement est simplifiée par la suppression de la substitution, celle-ci étant traitée par une nouvelle règle pour l'affectation inconditionnelle. Afin de simplifier les notations, dans ces règles les programmes sont tous des sous-programmes d'un programme régularisé (i.e. résultat d'une transformation syntaxique), et la profondeur est implicitement i. En particulier, les programmes sont réguliers, toutes les expressions

$$\frac{}{\{Q[B/\,Tmp_i], C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ Tmp_i \equiv B\ \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}}$$

$$\frac{\forall j \geq i, Tmp_j \notin Var(Q)}{\{Q[C \wedge \bigwedge_{k=0}^{i-1} Tmp_k ?E\colon X/X], C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ X := E\ \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}}$$

$$\frac{\begin{array}{c}\{P, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ S^*\ \{R, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \{R, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ T^*\ \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \forall j \geq i, Tmp_j \notin Var(R) \cup Var(Q)\end{array}}{\{P, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ S^*; \{R, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ T^*\ \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}}$$

$$\frac{\begin{array}{c}\{P, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \Rightarrow \{P', C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \{P', C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ S^*\ \{Q', C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \{Q', C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \Rightarrow \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \forall j \geq i, Tmp_j \notin Var(Q) \cup Var(Q')\end{array}}{\begin{array}{c}\{P, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \{P', C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ S^* \\ \{Q', C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\end{array}}$$

$$\frac{\begin{array}{c}\{P, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ S^*\ \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \forall j \geq i, Tmp_j \notin Var(Q)\end{array}}{\begin{array}{l}\{P, C \wedge \bigwedge_{k=0}^{i} Tmp_k\}\quad \textsf{where}\ \ B\ \textsf{do} \\ \qquad\qquad\qquad\qquad \{P, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \qquad\qquad\qquad\qquad S^* \\ \qquad\qquad\qquad\qquad \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\} \\ \qquad\qquad\qquad\qquad \textsf{end}\ \{Q, C \wedge \bigwedge_{k=0}^{i} Tmp_k\}\end{array}}$$

$$\frac{\{P, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ S^*\ \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}}{\{P, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}\ S^{**}\ \{Q, C \wedge \bigwedge_{k=0}^{i-1} Tmp_k\}}$$

où S^{**} est obtenu à partir de S^* par suppression d'une assertion quelconque.

FIG. 5.2 – *Règles d'annotations*

de conditionnement qui apparaissent sont des variables auxiliaires, et le contexte à la profondeur i est dénoté par $C \wedge \bigwedge_{k=0}^{i-1} Tmp_{i-1}$, où C est le contexte initial dans lequel le programme total est exécuté. *Les règles s'appliquent à des spécifications simples, i.e.* $Var(C) \cap Change(S) = \emptyset$.

Nous pouvons alors énoncer l'équivalence suivante.

Théorème 7 *Soit S un programme linéaire, et $\{P, C\}\ S\ \{Q, D\}$ une spécification simple telle que $\forall i,\ Tmp_i \notin Var(Q) \cup Var(D)$. Alors*

$$\{P, D\}\ (\overline{S})^*\ \{Q, D\} \textit{ est une preuve par annotations de } \overline{S}$$

$$\Updownarrow$$

$$\vdash^* \{P, D\}\ S\ \{Q, D\}$$

Preuve. *La preuve est identique à celle concernant la première méthode de preuve par annotations. Dans le sens direct, il faut seulement s'assurer par un argument sur la profondeur de conditionnement que la règle d'affectation inconditionnelle n'« écrase » jamais une variable auxiliaire utile. La réciproque fait appel au calcul des plus faibles préconditions, et utilise le fait que si $Tmp_i \notin Var(D)$, alors*

$$wp(Tmp_i \equiv B, \{Q, D\}) = \{Q[B/Tmp_i], D\}.$$

□

5.3 Programmes avec boucles

Nous nous sommes restreints dans notre étude aux programmes linéaires, c'est-à-dire sans boucles. En effet, le traitement des boucles requiert la mise en œuvre d'une machinerie complexe. La preuve d'une boucle repose sur l'existence d'un invariant. Dans le cas des langages scalaires, il est possible de prouver l'existence de cet invariant pour toute boucle à l'intérieur d'une spécification valide. Cette preuve repose sur la construction par un codage approprié de la plus faible précondition d'une boucle. Malheureusement, le calcul des plus faibles préconditions dont nous disposons pour le langage des assertions en deux parties ne le permet pas.

Cependant, il est possible de définir une méthode de preuves par annotations pour les boucles. Il nous suffit d'ajouter les deux règles suivantes. La première concerne le mécanisme d'étiquetage[3], et la seconde l'insertion d'annotations.

$$\varphi(\mathsf{loop}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end}, i) = (i)\,\mathsf{loop}\ \ B\ \ \mathsf{do}\ \ \varphi(S, i)\ \ \mathsf{end}$$

3. Rappelons que l'itération ne modifie pas le contexte.

$$\frac{\begin{array}{c}\{I \wedge \exists u : ((\bigwedge_{k=0}^{i} Tmp_i)|_u \wedge B|_u)\}\ S^*\ \{I\} \\ Lab(S) = i \\ \forall j > i,\ Tmp_j \notin Var(I)\end{array}}{\begin{array}{l}\{I\}\ (i)\ \mathsf{loop}\ \ B\ \ \mathsf{do} \\ \qquad \{I \wedge \exists u : ((\bigwedge_{k=0}^{i} Tmp_i)|_u \wedge B|_u)\} \\ \qquad S^* \\ \qquad \{I\} \\ \qquad \mathsf{end}\ \{I \wedge \forall u : ((\bigwedge_{k=0}^{i} Tmp_i)|_u \Rightarrow \neg B|_u)\}\end{array}}$$

Nous pouvons alors établir la propriété suivante.

Proposition 8 *Soit $\{P\}\ (0)S\ \{Q\}$ une formule où S est un programme éventuellement non linéaire, et telle que $\forall j,\ Tmp_j \notin Var(Q)$. Alors*

$$\{P\}\ S^*\ \{Q\}\ \textit{est une annotation pour } S$$

$$\Downarrow$$

$$\vdash^* \{P, Tmp_0\}\ S\ \{Q, Tmp_0\}$$

Nous ne montrerons pas ici la réciproque. Celle-ci repose en effet sur le calcul des plus faibles préconditions, que nous n'avons pas développé pour les boucles dans ce cadre. Celui-ci sera traité dans le cadre du chapitre 6

Preuve. *Comme pour les programmes linéaires, la preuve se fait par induction sur la longueur de la dérivation de la preuve par annotations. Nous n'avons donc qu'à vérifier le cas où la dernière règle appliquée était celle de la boucle, les autres cas ayant déjà été traités. Nous prouvons en fait le cas plus général où le programme S est étiqueté par i. Si la dernière règle appliquée était celle de la boucle, nous avons par hypothèse d'induction*

$$\vdash \{I \wedge \exists u : ((\bigwedge_{k=0}^{i} Tmp_i)|_u \wedge B|_u), \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{I, \bigwedge_{k=0}^{i} Tmp_i\}$$

La règle de la boucle du système de preuve s'applique et donne directement le résultat voulu. ❑

5.4 Discussion

Nous avons donc défini une notion de preuves par annotations pour le langage $\mathcal{L}$. Grâce à la séparation en deux parties des assertions, la méthode de validation proposée fonctionne en deux temps. La première phase consiste en un étiquetage dans le sens direct, la seconde génère les annotations en partant de la postcondition finale.

La première phase se résume à une simple réécriture, et doit uniquement tenir compte de la numérotation des variables temporaires. La seconde phase est similaire aux méthodes de preuves par annotations pour les langages scalaires de type Pascal. La seule différence provient d'un mécanisme de substitution légèrement plus complexe. Cette similarité confirme le fait que la validation

des programmes data-parallèles est du même niveau de « complexité » que celle des programmes scalaires. Ceci diffère fortement du cas des langages à parallélisme de contrôle du type CSP. Les langages data-parallèles apparaissent donc comme un modèle de programmation « raisonnable ».

Cette méthodologie de preuve par annotations pourrait être facilement automatisée et intégrée dans un outil d'aide à la conception et à la validation. La difficulté principale réside dans la complexité du mécanisme de substitution qui génère des assertions peu lisibles. Ces assertions devraient être simplifiées au fur et à mesure pour être compréhensibles par un utilisateur. L'intervention de l'utilisateur est en effet indispensable pour fournir les invariants de boucle. D'autres optimisations pourraient être mises en œuvre, par exemple lorsque les expressions de contexte ne sont pas modifiées par les instructions. Il ne serait pas nécessaire dans ce cas d'introduire de variable auxiliaire, ce qui procurerait un gain de lisibilité. Nous ne nous sommes pas attachés à fournir un tel outil de preuve semi-automatique. Le langage $\mathcal{L}$ est en effet un langage d'étude et un prototype utilisant celui-ci serait de peu d'intérêt au niveau pratique. En revanche, cette étude, ainsi que d'autres travaux sur des extensions de $\mathcal{L}$ effectués par Bougé et Levaire ou Le Guyadec et Virot [12, 46], montre qu'il serait possible de mettre en œuvre de tels outils pour des langages existants dont $\mathcal{L}$ est inspiré.

Le langage d'assertion utilisé ici est la base d'une méthode simple de validation. Cependant, il souffre d'insuffisances du point de vue théorique. Nous allons voir dans le chapitre suivant comment un autre langage d'assertion permet de pallier à ces insuffisances.

chapitre 6

Un autre langage d'assertion[1]

Les assertions en deux parties présentent l'avantage d'une lecture simple et intuitive. Si l'on introduit des restrictions appropriées, il est possible de les utiliser à un niveau pratique, étant donné que nous sommes capables de garantir la validité d'une preuve par annotations. Cependant, nous avons vu dans le chapitre 4 que le langage des assertions en deux parties n'était pas suffisamment expressif pour établir la complétude d'un système de preuve. Ces assertions s'avèrent donc insuffisantes du point de vue théorique. Nous allons revenir ici sur le problème de la définissabilité des plus faibles préconditions exprimées dans ce langage. Ceci nous amènera à définir un autre langage d'assertion, dont l'intuition est moins immédiate, mais qui est plus expressif.

6.1 Retour sur la définissabilité

Nous avons vu au début du chapitre 4 que le langage d'assertion utilisé jusqu'ici n'a pas la propriété de définissabilité, c'est-à-dire que nous ne sommes pas capables d'exprimer les plus faibles préconditions d'une paire quelconque (programme, assertion) sous forme d'une assertion. Nous avons vu plus précisément que ce problème venait du fait que le contexte ne pouvait prendre qu'une valeur dans un environnement donné. Nous rappelons ce fait essentiel :

Fait (Expressivité réduite des assertions) *Soit $\{P, C\}$ une assertion. Pour tout environnement σ, il existe au plus un contexte c tel que $(\sigma, c) \models \{P, C\}$: précisément, $c = \sigma(C)$.*

Nous avons montré que certains cas pouvaient néanmoins être traités. L'introduction de variables auxiliaires permet en effet de s'affranchir des problèmes induits par les interactions entre les variables des assertions ou des expressions de conditionnement, et celles modifiées par les programmes. Reste un cas non résolu, celui des boucles. Nous avons en effet donné un exemple où la divergence était un obstacle à la définissabilité.

Nous pourrions adopter la « tactique » suivante : les problèmes viennent des plus faibles préconditions des boucles ; or nous savons que pour les langages scalaires, l'expression de ces préconditions se fait par un codage complexe. Ce codage est utile pour prouver la complétude, mais n'est jamais mis en œuvre en pratique. L'utilisation réelle d'un système de preuve nécessite l'intervention de l'utilisateur, qui fournit lui même l'invariant de boucle sous forme d'une assertion. En étant optimiste, nous pourrions penser que dans le cas présent, il est possible de donner un tel invariant. Cet invariant ne serait pas obligatoirement le plus faible, mais serait exprimable par une assertion.

1. La matière de ce chapitre est tirée de *Proving data-parallel programs: a unifying approach* (D. Cachera et G. Utard, [16]).

Il permettrait d'énoncer rapidement un résultat de complétude faible, du type : « s'il existe un invariant exprimable sous forme d'assertion pour toute boucle, alors il est possible de prouver la spécification voulue. »

Pourtant, même cette simplification, qui repose sur une hypothèse forte, n'est pas possible. En effet, l'exemple suivant nous montre que la divergence d'une boucle peut être « cachée » par un mécanisme de conditionnement. Considérons le programme S suivant, sur un domaine monodimensionnel $\mathcal{D} = [1..M]$.

$X := False;$
where $This = 1$ do
    $N := 1; X := True;$
    loop $X|_N$ do
        $N := N + 1$
    end
end

Intuitivement, la valeur de $X|_u$ est initialisée à *false pour chaque indice actif*. Ensuite, les valeurs de $X|_1$, $X|_2$, etc. sont lues successivement jusqu'à ce que la valeur *false* soit trouvée. Rappelons que nous supposons que la fonction stoi est définie partout, donc que ces communications ont un sens (considérons par exemple une indexation cyclique comme en MPL).

Posons

$$P \equiv (\forall u \neq 1 : X|_u = true) \qquad \text{et} \qquad C \equiv (This = 1)$$

Nous avons alors le fait suivant.

Fait *Le programme S diverge à partir de (σ, c) ssi $(\sigma, c) \in [\![\{P, C\}]\!]$.*

Preuve. *S diverge si et seulement si l'indice 1 est actif en entrée de la boucle et qu'aucun $X|_u$ n'est faux. Du fait de l'initialisation $X := False$, $X|_u$ est faux si et seulement si l'indice u est actif initialement. Les indices $u \neq 1$ doivent donc être inactifs initialement pour qu'il y ait divergence. Le contexte initial doit donc être décrit par $This = 1$. De plus, chaque $X|_u$ doit être vrai initialement, sauf pour $u = 1$. Réciproquement, ces conditions sont suffisantes.* ❑

Une première obervation est que la valeur de C ne dépend pas de l'environnement. Nous avons alors

Fait *wlp(S,$\{true, C\}$)=$\{true, C\}$.*

Preuve. *Soit $(\sigma, c) \in [\![\{true, C\}]\!]$. Si S diverge à partir de cet état, il appartient par définition à la plus faible précondition. Si S converge, alors il y a conservation du contexte et l'état final valide bien la postcondition.*

Réciproquement, soit s un état tel que $[\![S]\!](s) \in [\![\{true, C\}]\!]$. Si S diverge à partir de cet état, alors d'après le fait précédent $s \in [\![\{P, C\}]\!] \subseteq [\![\{true, C\}]\!]$. S'il converge, alors le contexte est toujours décrit par C, puisque C ne dépend pas de l'environnement. ❑

Nous sommes donc capable d'exprimer la plus faible précondition de S et de $\{true, C\}$ par une assertion. En revanche, cela est impossible pour where C do S end, comme le montre le fait suivant.

Fait *wlp(*where C do S end,$\{true, True\}$) n'est pas exprimable par une assertion.*

Preuve. *Fixons un environnement $\sigma \models P$. Alors à la fois$(\sigma, True)$ et $(\sigma, \sigma(C))$ appartiennent à $wlp(\text{where } C \text{ do } S \text{ end}, \{true, True\})$. Comme $\sigma(C) \neq True$, cet ensemble ne peut être décrit par une assertion.* ❑

De cet exemple, nous pouvons conclure qu'il est indispensable de définir un langage d'assertion plus « riche », où le contexte ne dépende pas fonctionnellement de l'environnement.

6.2 Un nouveau langage

Dans [10], Bougé et al. établissent le résultat suivant.

Théorème 8 *Soit $\{Q, D\}$ une assertion, et S un programme linéaire. Soit Aux une nouvelle variable n'apparaissant pas dans $Var(Q) \cup Var(D) \cup Var(S)$. Il existe un prédicat $P(Aux)$ tel que*

$$wp(S, \{Q, D\}) = \{(\sigma, c) \mid \sigma[Aux \leftarrow c] \models P(Aux)\}$$

Ce résultat suggère une nouvelle voie pour la définition du langage d'assertion. Nous avons en effet jusqu'à maintenant exploré deux voies : l'une consiste à introduire des variables auxiliaires dans les règles de preuve ; l'autre introduit ces variables directement dans le programme, par transformation syntaxique. Ce que nous proposons ici, c'est d'introduire l'équivalent de ces variables auxiliaires directement dans les assertions.

6.2.1 Assertions

Une assertion est une paire $\langle W, A\rangle$ où W est un prédicat sur les variables de programme, et A est une variable parallèle qui s'évalue en le contexte courant. La structure des prédicats est la même que pour les assertions en deux parties précédentes. La variable A peut naturellement apparaître dans le prédicat.

La définition de la satisfiabilité d'une assertion est modifiée comme suit.

Définition 12 (Satisfiabilité) *L'état (σ, c) satisfait l'assertion $\langle W, A\rangle$ si et seulement si*

$$\sigma[A \leftarrow c] \models W$$

De même que précédemment, un mécanisme de substitution est introduit, qui vérifie les propriétés usuelles.

6.2.2 A-conversion

Dans la définition de la satisfiabilité, la variable A nous permet de décrire des propriétés du contexte dans le prédicat W. La valeur initiale de A n'a aucune importance, puisqu'elle est écrasée dans la substitution $\sigma[A \leftarrow c]$. Cette variable joue en quelque sorte le rôle d'une variable muette. Nous introduisons un mécanisme de A-conversion qui nous permet de renommer cette variable. Ce mécanisme est semblable à l'α-conversion du lambda-calcul.

Proposition 9 (A-conversion) *Soit W un prédicat et A' tel que $A' \notin Var(W)$. Nous avons*

$$[\![\langle W, A\rangle]\!] = [\![\langle W[A'/A], A'\rangle]\!]$$

Preuve. *Soit* $(\sigma, c) \in [\![\langle W, A\rangle]\!]$. *Par définition,* $\sigma[A \leftarrow c] \models W$. *Comme* $A' \notin Var(W)$, $\sigma[A \leftarrow c][A' \leftarrow c] \models W$. *Notons* $\sigma[A' \leftarrow c]$ *par* σ' *Nous avons donc* $\sigma'[A \leftarrow \sigma'(A')] \models W$. *Par le lemme de substitution, nous concluons que* $\sigma[A' \leftarrow c] \models W[A'/A]$, *c'est-à-dire* $(\sigma, c) \in [\![\langle W[A'/A], A'\rangle]\!]$.

Réciproquement, soit $(\sigma, c) \in [\![\langle W[A'/A], A'\rangle]\!]$. *Nous avons* $\sigma[A' \leftarrow c] \models W[A'/A]$. *Par le lemme de substitution, nous avons* $\sigma'[A \leftarrow \sigma'(A')] \models W$, *donc* $\sigma[A \leftarrow c][A' \leftarrow c] \models W$. *Comme* $A' \notin Var(W)$, *nous concluons que* $\sigma[A \leftarrow c] \models W$. ❑

6.2.3 Implication

La définition de l'implication est modifiée, afin de tenir compte de la A-conversion.

Définition 13 (Implication des assertions) *Soit* $\langle W, A\rangle$ *et* $\langle W', A'\rangle$ *deux assertions. On a l'implication* $\langle W, A\rangle \Rightarrow \langle W', A'\rangle$ *dans les deux cas suivants*

- *A est la même variable que* A' *et* $W \Rightarrow W'$ *dans le sens où* $\forall \sigma : \sigma \models W \Rightarrow \sigma \models W'$;
- *A est distinct de* A' *et* $W[A''/A] \Rightarrow W'[A''/A']$, *avec* $A'' \notin Var(W) \cup Var(W')$.

Remarquons que le second cas correspond à une A-conversion simultanée des deux assertions et que A'' est une nouvelle variable n'apparaissant ni dans W ni dans W'. Il est toujours possible de trouver une telle variable, car toutes les expressions de notre langage sont des termes finis. Remarquons aussi que si A n'apparaît pas dans W' (respectivement A' dans W), choisir $A'' = A$ (respectivement $A'' = A'$) est correct et ne nécessite qu'une substitution.

La définition de l'implication respecte naturellement la propriété

$$\langle W, A\rangle \Rightarrow \langle W', A'\rangle \qquad \Leftrightarrow \qquad [\![\langle W, A\rangle]\!] \subseteq [\![[\![\langle W', A'\rangle]\!]]\!]$$

La validité d'une spécification est définie de la même façon que pour le langage d'assertion précédent.

Nous pouvons alors donner un nouveau système de preuve adapté à ce type d'assertion. Grâce à la A-conversion, nous pouvons supposer que la variable de contexte A n'est jamais dans $Var(S)$. Le système de preuve est donné en figure 6.1. Il est bien entendu correct : toute spécification prouvable est valide.

6.3 Complétude

Nous allons maintenant être capable de prouver la complétude de notre système de preuve. En effet, le langage d'assertion introduit dans ce chapitre a la propriété de définissabilité. Nous redonnons tout d'abord la définition des plus faibles préconditions et l'énoncé du lemme de conséquence.

Définition 14 (Plus faibles préconditions) *Soit* S *un programme et* $\mathcal{E}$ *un sous-ensemble de Etats.*

$$wlp(S, \mathcal{E}) = \{(\sigma, c) \mid [\![S]\!](\sigma, c) \in \mathcal{E}\}$$

Lemme 4 (Conséquence) $\models \langle V, A'\rangle\ S\ \langle W, A\rangle$ *ssi* $[\![\langle V, A'\rangle]\!] \subseteq wlp(S, \langle W, A\rangle)$.

Nous avons alors la propriété de définissabilité suivante.

Proposition 10 (Définissabilité) *Soit* S *un programme, et* $\langle W, A\rangle$ *une assertion. Il existe une assertion* $\langle V, A'\rangle$ *telle que*

$$[\![\langle V, A'\rangle]\!] = wlp(S, [\![\langle W, A\rangle]\!]).$$

$$\frac{}{\langle W[A?E\colon X/X], A\rangle\ X{:=}E\ \langle W, A\rangle} \qquad \text{(Affectation)}$$

$$\frac{\langle W, A\rangle\ S\ \langle V, A\rangle \quad \langle V, A\rangle\ T\ \langle U, A\rangle}{\langle W, A\rangle\ S\ ;\ T\ \langle U, A\rangle} \qquad \text{(Séquencement)}$$

$$\frac{\langle W, A'\rangle\ S\ \langle V, A'\rangle, A' \notin Var(V)}{\langle W[A \wedge B/A'], A\rangle\ \mathsf{where}\ B\ \mathsf{do}\ S\ \langle V, A\rangle} \qquad \text{(Conditionnement)}$$

$$\frac{\langle I \wedge \exists u : (A|_u \Rightarrow B|_u), A\rangle\ S\ \langle I, A\rangle}{\langle I, A\rangle\ \mathsf{while}\ B\ \mathsf{do}\ S\ \langle I \wedge \forall u : (A|_u \Rightarrow \neg B|_u), A\rangle} \qquad \text{(Itération)}$$

$$\frac{\langle W', A'\rangle \Rightarrow \langle W, A\rangle \quad \langle W, A\rangle\ S\ \langle V, A\rangle \quad \langle V, A\rangle \Rightarrow \langle V', A''\rangle}{\langle W', A'\rangle\ S\ \langle V', A'\rangle} \qquad \text{(Conséquence)}$$

FIG. 6.1 – *Système de preuve*

La preuve de cette proposition repose sur les lemmes suivants, dont nous donnons les preuves en annexe A. Le fait que la variable de contexte de la précondition (A') est différente de celle de la postcondition (A) vient d'éventuelles A-conversions nécessaires si la variable de contexte est modifiée par une affectation. Les lemmes concernant l'affectation et le conditionnement sont très simples. Celui concernant l'itération est plus complexe. Il faut en effet décrire un codage des états par les entiers, afin de pouvoir manipuler ceux-ci dans des formules arithmétiques du langage d'assertion. Nous donnons en annexe une idée de la preuve et une justification de ses arguments, sans entrer dans les détails du codage.

Lemme 5 (Affectation) *Soit X une variable, E une expression vectorielle, $\langle W, A\rangle$ une assertion. Il existe une assertion $\langle V, A'\rangle$ telle que*

$$wlp(X{:=}E, \langle W, A\rangle) = [[\langle V, A'\rangle]] .$$

Lemme 6 (Conditionnement) *Soit S un programme, B une expression vectorielle booléenne, $\langle W, A\rangle$ une assertion telle que $C \notin Var(B) \cup Change(S)$, et A' une variable telle que $A' \notin Var(W)$. S'il existe un prédicat W' tel que*

$$wlp(S, \langle W, A'\rangle) = [[\langle W', A'\rangle]],$$

alors

$$wlp(\mathsf{where}\ B\ \mathsf{do}\ S, \langle W, A\rangle) = [[\langle W'[C \wedge B/A'], A\rangle]] .$$

Lemme 7 (Itération) *Soit T un programme et $\langle W, A\rangle$ une assertion. Il existe une assertion $\langle V, A'\rangle$ telle que*

$$[[\langle V, A'\rangle]] = wlp(\mathsf{loop}\ B\ \mathsf{do}\ T, [\![\langle W, A\rangle]\!]).$$

Nous ne détaillerons pas la preuve de la proposition 10. Celle-ci se fait classiquement, par induction sur la structure du programme. Il faut simplement choisir des variables de contexte appropriées, et procéder à des A-conversions lorsque cela s'avère nécessaire du fait des affectations.

Un dernier lemme technique est nécessaire pour prouver la complétude. Ce lemme exprime le fait que si une variable n'apparaît ni dans le programme ni dans la postcondition, alors il est toujours possible d'exprimer la plus faible précondition par une assertion ne contenant pas non plus cette variable.

Lemme 8 (Propagation) *Soit S un programme et $\langle W, A\rangle$ une assertion. Si X est une variable telle que $X \notin Var(S) \cup Var(W) \cup \{A\}$, alors il existe une assertion $\langle V, A'\rangle$ telle que $[\![\langle V, A'\rangle]\!] = wlp(S, [\![\langle W, A\rangle]\!])$ et $X \notin Var(V) \cup \{A'\}$.*

La preuve de ce lemme suit exactement celle de la définissabilité, et consiste uniquement en un choix judicieux des variables de contexte.

Nous pouvons finalement énoncer le résultat de complétude. Il s'agit bien entendu de complétude relative, au sens de Cook [22][2]. Le résultat énoncé ici est plus complet que celui obtenu avec le langage des assertions en deux parties des chapitres précédents, car il inclut les programmes avec boucles.

Théorème 9 (Complétude) *Soit S un programme, et $\langle V, A'\rangle$ et $\langle W, A\rangle$ deux assertions. Si*

$$\models \langle V, A'\rangle\ S\ \langle W, A\rangle,$$

alors

$$\vdash \langle V, A'\rangle\ S\ \langle W, A\rangle.$$

Nous ne donnons pas ici la preuve de ce théorème, exactement identique à celles des théorèmes de complétude du chapitre 4. Les preuves détaillées des résultats énoncés dans ce chapitre peuvent être trouvées dans [16].

6.4 Bilan

Nous avons présenté dans ce chapitre un autre langage d'assertion qui résout de façon beaucoup plus élégante que les autres approches le problème de la définissabilité des plus faibles préconditions, et par là même celui de la complétude. Ceci se fait au détriment d'aspects plus pratiques, comme la possibilité d'effectuer des preuves par annotations en deux phases séparant clairement le traitement du contexte de celui des instructions. Ce point est discuté plus précisément dans le chapitre suivant, qui conclut la première partie de cette thèse.

2. Un « véritable » résultat de complétude exigerait un langage d'assertion lui-même complet, c'est-à-dire dont toutes les formules vraies soient axiomatiquement prouvables. La complétude au sens de Cook admet au rang d'axiomes toutes ces formules vraies.

chapitre 7

Conclusion

Nous avons présenté dans les premiers chapitres de cette thèse une méthode de validation formelle pour un langage data-parallèle impératif. Le langage choisi, quoique simple, est assez représentatif de nombreux langages data-parallèles « réels ». Comme pour les langages impératifs classiques, la méthode choisie consiste à tirer parti de la structure syntaxique du programme, pour insérer dans le texte de celui-ci des assertions qui doivent être vérifiées lorsque le contrôle atteint le point considéré. C'est justement la nature *data*-parallèle du langage, c'est-à-dire en l'occurrence l'unicité du flot de contrôle, qui permet de développer une telle approche.

Nous avons passé en revue au cours de cette thèse plusieurs approches voisines de ce problème. Ces approches diffèrent dans la façon dont sont traités les problèmes liés au contexte d'activité. On remarquera que, quelle que soit la stratégie adoptée, celle-ci fait appel à des variables auxiliaires et à des substitutions.

- La première voie dans laquelle nous nous sommes engagés propose de placer ces substitutions au niveau des règles de preuve. Cette approche, qui prolonge les travaux antérieurs de Bougé et al., privilégie la simplicité. Les assertions en deux parties sont en effet claires et correspondent à l'intuition, puisqu'elles suivent de près la syntaxe. De plus, la séparation du prédicat portant sur les variables du programme et de la partie contexte nous a permis de définir une méthodologie simple de preuve par annotations, se rapprochant du cas impératif scalaire. Même si des preuves par annotations étaient déjà présentes dans certains travaux antérieurs, elles l'étaient en l'absence de tout cadre formel justifiant le bien-fondé de leur utilisation. Nous avons ici justifié et précisé une méthodologie concrète. Cette première voie souffre cependant d'insuffisances théoriques dues au manque d'expressivité du langage d'assertion en ce qui concerne le contexte. Néanmoins, nous pouvons supposer que cette approche reste utilisable dans des cas réels. En effet, le point délicat à traiter dans une preuve par annotations d'un programme écrit dans un langage scalaire est de fournir des invariants corrects pour les boucles. Ceci nécessite l'intervention de l'utilisateur, et nous n'y échapperons pas dans le cas data-parallèle.

- Parallèlement à cette première approche, nous avons mis en évidence une seconde méthode qui consiste à opérer des transformations sur le texte du programme. Le problème des substitutions est donc ici géré au niveau de la syntaxe, de façon immédiatement visible par l'utilisateur. Cette seconde méthode permet de façon immédiate de définir un cadre de preuves par annotations identique à celui de la première approche.

- Une autre voie a consisté en une modification du langage d'assertion. Dans ce dernier cas, le problème de la substitution est transféré au niveau des assertions, par le mécanisme des

A-conversions. L'idée était en germe dans la thèse de Utard au niveau du calcul des plus faibles préconditions. Nous l'avons développée au cours d'un travail commun, définissant le mécanisme de conversion, le système de preuve et établissant la preuve de complétude. Cette dernière voie résout les problèmes théoriques rencontrés précédemment. En revanche, elle ne permet plus de séparer en deux phases le traitement du contexte et celui des prédicats. Il serait possible de définir des preuves par annotations reposant sur ce second langage. Nous ne l'avons pas fait ici, la démarche étant en presque tous points identique à celle utilisée pour le premier langage. Une telle méthodologie fournirait éventuellement des preuves en une seule phase, mais moins claires au niveau de la lecture des annotations.

Ces approches présentent toutes un bilan nuancé. Certaines privilégient l'intuition, d'autres l'élégance au niveau de la théorie. Un compromis est à trouver entre aspects pratiques et complétude de la méthode lors d'une éventuelle mise en œuvre. D'autre part, la nécessité de procéder à des substitutions pour traiter le problème du contexte d'activité est probablement à rapprocher des méthodes de preuve utilisées pour certains langages à parallélisme de contrôle, où l'introduction de variables auxiliaires est nécessaire pour disposer d'informations sur le flot de contrôle (cf. Apt et Olderog, [1]). Ceci semble être le prix à payer pour passer du cas scalaire au cas data-parallèle.

Lien avec d'autres travaux

Les travaux visant à équiper le langage $\mathcal{L}$ d'un système de preuve ont débuté avec le langage des assertions en deux parties exposé au chapitre 2. Devant les insuffisances de ce langage d'assertion, plusieurs approches ont été proposées. Nous en avons exposé deux, l'une consistant à garder le même langage d'assertion et à introduire des variables auxiliaires, l'autre consistant à définir un autre langage d'assertion.

Le Guyadec et Virot [47] ont proposé une approche légèrement différente pour traiter ce problème. Ils définissent un langage où les assertions sont en une partie, et modifient légèrement la sémantique opérationnelle du langage. Dans cette approche, une assertion est simplement un prédicat, où le contexte est dénoté par une variable réservée notée $\sharp$. Cette variable ne peut apparaître dans les programmes. En ce qui concerne la sémantique opérationnelle, la partie contexte des états est supprimée. Un état consiste seulement en un environnement, et le contexte est dénoté par la valeur de $\sharp$ dans cet environnement. Tout environnement dont le support contient la variable $\sharp$ peut donc définir un état. Notons $[\![S]\!]_{GV}$ la sémantique d'un programme dans cette approche. Nous allons mettre en évidence le lien qui existe entre la méthode de Le Guyadec et Virot et celle fournie par les assertions $\langle W, A\rangle$.

Tout d'abord, examinons le côté opérationnel. Sans vouloir en faire ici la preuve formelle, nous pouvons affirmer le point suivant :

$$[\![S]\!]_{GV}(\sigma) = \sigma' \quad \Rightarrow \quad [\![S]\!](\sigma, c) = (\sigma', c) \text{ avec } c = \sigma(\sharp)$$

et

$$[\![S]\!](\sigma, c) = (\sigma', c) \quad \Rightarrow \quad [\![S]\!]_{GV}(\sigma[\sharp \leftarrow c]) = \sigma'[\sharp \leftarrow c]$$

Les comportements des programmes sont donc identiques.

Examinons ensuite le langage d'assertion. Le langage défini par Le Guyadec et Virot leur a permis de définir un calcul des plus faibles préconditions ayant la propriété de définissabilité. En fait, il existe une relation très simple entre les deux types d'assertions : à toute assertion $\langle W, A\rangle$ telle que $\sharp \notin Var(W)$, on peut faire correspondre une assertion $\langle W[\sharp/A]\rangle_{GV}$. Ce langage est donc

une restriction de la forme générale $\langle W, A\rangle$. La différence principale vient du fait qu'il introduit un aspect opérationnel (la variable $\sharp$, qui représente le contexte de façon « câblée ») dans le langage logique.

Les deux dernières approches que nous avons présentées suggèrent de revenir un instant sur l'origine profonde du manque d'expressivité du langage des assertions en deux parties initial. En effet, l'argument avancé était que pour un environnement donné, il ne peut y avoir qu'un seul contexte. L'approche de Le Guyadec et Virot donne des résultats alors que dans un environnement σ il ne peut toujours y avoir qu'un contexte : $\sigma(\sharp)$. En fait, le langage d'assertion doit permettre d'exprimer *suffisamment d'information* sur le contexte.

Dans l'exemple du programme $S \equiv X := X + 1$ et de la postcondition $\{(\forall u : X|_u = 1) \vee (\forall u : X|_u = 2), (X = 2)\}$ donné en page 46 nous devons être en mesure d'exprimer une disjonction sur le contexte. Dans l'exemple du programme

$$\mathsf{loop}\ \mathit{True}\ \mathsf{do}\ X := X\ \mathsf{end}$$

associé à la postcondition $\{\mathit{true}, \mathit{True}\}$, nous devons être en mesure d'exprimer une propriété de nature opérationnelle (il existe un processeur actif) dans la logique. Le Guyadec et Virot y parviennent en introduisant le contexte, fixé dans une variable réservée, dans les environnements. Notre approche englobe cette dernière et évite cet artifice.

Critiques et perspectives

Le travail portant sur le langage $\mathcal{L}$ ne peut prétendre à une vision générale du problème de la validation des langages à parallélisme de données. Tout d'abord, par la nature même du langage étudié, le modèle présenté est purement impératif. le langage $\mathcal{L}$ ne permet d'exprimer que les constructions de base du modèle data-parallèle, et constitue donc en quelque sorte un « assembleur abstrait data-parallèle », de bas niveau mais dégagé de toute considération d'architecture ou de machine-cible. Il serait intéressant d'étendre le travail à des modèles de plus haut-niveau, et pas nécessairement impératifs. Nous abordons dans la seconde partie de cette thèse un langage de nature davantage fonctionnelle. L'ensemble constitue néanmoins un vaste sujet d'étude.

Même si l'on se cantonne au modèle impératif, le langage $\mathcal{L}$ souffre d'une autre limitation : il ne traite que des aspects liés au contrôle, sans se soucier des problèmes liés aux données. Cette limitation ne contredit pas le but initial de notre étude, qui était de montrer la similitude entre un langage impératif scalaire et un langage impératif data-parallèle. Il serait néanmoins intéressant de pouvoir traiter les problèmes liés aux données, que ceux-ci soient des problèmes de placement — et donc de coût des communications — ou des problèmes d'échelle. Ici encore, un vaste sujet détude s'ouvre devant nous.

Finalement se pose la question de l'intégration des méthodes de validation proposées dans un outil automatique d'assistance à la preuve. Des premiers essais ont été effectués par Levaire [50], puis Mounier et Utard [53] au début des travaux sur $\mathcal{L}$. Ces essais utilisaient l'atelier sémantique Centaur. Ils font apparaître qu'une semi-automatisation serait possible, mais également que les assertions générées sont très complexes du fait des substitutions conditionnées par le contexte. Un véritable outil devrait donc comporter des stratégies spécifiques pour simplifier au fur et à mesure ces substitutions. Un avantage certain du modèle data-parallèle par rapport au modèle à parallélisme reste néanmoins l'absence d'explosion combinatoire de la complexité des preuves en fonction de la taille du problème à traiter.

Deuxième partie

Le langage ALPHA

chapitre 1

Introduction

Dans la première partie de cette thèse, nous avons étudié un langage data-parallèle de nature strictement impérative. Nous allons maintenant nous intéresser à un langage d'un autre type. Le langage lui-même est présenté dans le chapitre suivant, nous n'en mentionnons ici que les caractéristiques générales afin d'exposer notre but. Le langage ALPHA est développé à l'IRISA (Rennes) au sein du projet API (Architectures Parallèles Intégrées). Ce projet s'intéresse aux architectures spécialisées VLSI, et en particulier aux architectures parallèles, fréquemment utilisées en traitement du signal, de l'image, etc. Les applications possibles sont par exemple la comparaison de séquences biologiques, la compression d'image, ou le codage et le cryptage de données. Dans cette optique, le langage ALPHA a été conçu pour la spécification et la synthèse de circuits systoliques, qui représentent une classe particulière d'architecture parallèle régulière.

Le langage ALPHA est fondé sur le formalisme des *équations récurrentes affines*. Sa nature équationnelle le classe donc parmi les langages de type fonctionnel. Il peut être vu également comme un langage de type « flot de données » (*dataflow*). C'est aussi un langage data-parallèle, qui n'opère plus sur des vecteurs ou des tableaux, mais sur des domaines polyédriques. Nous verrons également qu'une caractéristique importante de ce langage est d'être à *assignation unique*, c'est-à-dire que les variables ne peuvent être affectées qu'une seule fois.

Nous avons mentionné dans l'introduction de cette thèse que l'approche interne était bien développée en ALPHA. En effet, le système ALPHA lui-même, tel qu'il est actuellement développé, repose une chaîne de transformations, qui sont en fait des réécritures. L'environnement implanté, conçu comme une sur-couche du système MATHEMATICA, permet d'opérer des transformations et des raffinements tels que l'ajout de variables intermédiaires, la sérialisation des communications, l'ordonnancement et le changement de base en espace-temps, etc. [30, 61]. Cette chaîne de transformations permettra à terme de passer d'une spécification équationnelle de haut niveau à une description de circuit. Toutes ces transformations — dès lors que leur correction a été prouvée — préservent la sémantique des programmes. Il est donc possible de prouver l'équivalence de programmes ALPHA directement dans le système ALPHA, par exemple par transformations successives aboutissant à une forme commune.

Cependant, malgré la richesse de ces transformations internes, de nombreux besoins ne sont pas couverts au niveau de la validation. L'étude de la littérature ainsi que des discussions avec les membres de l'équipe développant ALPHA ont fait apparaître plusieurs types de problèmes.

- Au niveau des spécifications : même si l'on prouve que toutes les transformations implantées en MATHEMATICA sont correctes, ceci ne permet d'établir que l'équivalence de la spécification initiale avec la forme finale. Lorsque les spécifications deviennent quelque peu complexes, il

devient nécessaire de vérifier que celles-ci respectent certaines propriétés. Ceci devient crucial lors de la conception modulaire de systèmes.

- Les propriétés que l'on peut souhaiter prouver ne concernent pas nécessairement toute la sémantique du programme. Nous appelons ce type de propriétés des propriétés *partielles*. Dans ce cas, il semble inadapté de raisonner par équivalence.

- Il apparaît que le langage ALPHA lui-même est trop faible pour exprimer les propriétés désirées. Ceci est dû notamment à la forme trop restreinte des domaines polyédriques, ainsi qu'à l'impossibilité d'exprimer des propriétés inductives.

Nous allons ici illustrer ces différents points sur des exemples, et montrer que, alors que le langage ALPHA par nature favorise une approche de type interne, une approche externe s'avère également nécessaire. Nous ne donnons pas auparavant une description complète du langage, celle-ci peut se trouver au chapitre suivant. Signalons seulement qu'un programme est constitué d'un système, comprenant des variables d'entrée déclarées après le nom du système, des variables de sortie déclarées après le mot clef `return`, des variables locales déclarées après le mot clef `var`, et un ensemble d'équations compris entre les mots-clés `let` et `tel`. Toutes les variables sont déclarées statiquement sur des domaines polyédriques. Les exemples donnés ici sont très simples et ne nécessitent pas une véritable connaissance du langage.

1.1 Propriétés d'équivalence

Ces propriétés sont des propriétés d'équivalence de programmes effectuant le même calcul — au sens fonctionnel — mais différant par la syntaxe ou par l'« algorithme » de calcul.

Une première catégorie parmi ces propriétés d'équivalence est celle des propriétés purement structurelles. Considérons par exemple le programme suivant, calculant une convolution sur une fenêtre de largeur 4.

```
system convolution (a : {j | 1<=j<=4} of integer;
                    x : {i | i>=1} of integer)
returns (y : {i | i>=4} of integer);
var
    Y : {i,j | 0<=j<=4 ; i>=4} of integer;
let
    Y[i,j] = case
            {| j=0 } : 0[];
            {| 1<=j<=4} : Y[i,j-1] + a[j] * x[i-j+1];
            esac;
    y[i] = Y[i,4];
tel;
```

Nous aimerions prouver que ce programme est équivalent à un autre, où la variable $Y[i,j]$ a été calculée de façon différente. Par exemple, si l'on « déroule » le calcul sur les premières valeurs de j :

```
Y[i,j] = case
            {| j=0 } : 0[];
            {| j=1 } : a[1] * x[i];
            {| 2<=j<=4} : a[j-1] + Y[i,j-1]
                    + a[j] * x[i-j+1] - a[j-1] ;
         esac;
```

Bien entendu, une équivalence de ce type peut être prouvée grâce à des règles de réécriture suffisamment puissantes. Remarquons cependant que de telles règles feraient intervenir des connaissances de nature sémantique, telles que $a + b + c - a = b + c$, qui ne sont pas toujours triviales (qu'en est-il par exemple dans le cas d'un dépassement de capacité?).

Un autre exemple, légèrement plus subtil, consiste à introduire une variable supplémentaire. Ce type de transformation est utilisé par exemple pour la sérialisation (pipeline) des communications afin d'obtenir un circuit systolique. Si l'on se restreint au cas le plus simple, cela peut par exemple donner, en introduisant la variable *aux* dans notre programme de convolution :

```
system convolution (a : {j | 1<=j<=4} of integer;
                    x : {i | i>=1} of integer)
returns (y : {i | i>=4} of integer);
var
    Y : {i,j | 0<=j<=4 ; i>=4} of integer;
    aux : {i | i>=4} of integer;
let
    Y[i,j] = case
            {| j=0 } : 0[];
            {| 1<=j<=4} : Y[i,j-1] + a[j] * x[i-j+1];
            esac;
    aux[i] = Y[i,4];
    y[i] = aux[i];
tel;
```

Là encore, un système de réécriture est capable de transformer les deux dernières équations en

```
y[i] = Y[i,4];
aux[i] = y[i];
```

Cependant, même si l'on est capable de détecter le fait que la variable *aux* est « inutile », a-t-on toujours le droit d'éliminer l'équation contenant cette variable? Et si cette équation était par exemple `aux[i] = y[i]/a;` avec a éventuellement nul? Ces exemples montrent en tout cas que les règles de réécriture doivent être maniées avec la plus grande prudence.

Une seconde catégorie parmi les propriétés d'équivalence est celle concernant les programmes effectuant les mêmes calculs, mais par des algorithmes différents. Cette fois, il n'est pas toujours

possible de raisonner par réécriture. C'est le cas notamment lorsque l'un des calculs se fait de façon récursive. Considérons par exemple le programme suivant.

```
system propagation : (a : {|} of integer)
returns (X : {t | t >= 0} of integer);
let
    X[t] = case
            {| t=0} : a[];
            {| t>0} : X[t-1];
        esac;
tel;
```

Ce programme exprime simplement une propagation de la constante a. Ce genre de propagation est largement utilisé dans les circuits systoliques. Naturellement, nous aimerions prouver que pour tout i, $X[i] = a$. Comme nous ne connaissons pas *a priori* la valeur de i, nous ne pourrons y parvenir par réécriture. Un autre exemple très proche consiste à remplacer dans le programme précédent `X[t-1]` par `X[t-1]+1`, et `a` par `0[]`. Nous avons alors défini un compteur[1]. Jusqu'à maintenant, la preuve que ce genre de définition satisfait la propriété voulue de propagation de constante ou d'implantation d'un compteur se faisait par induction « à la main », en l'absence de cadre formel pour la justifier (voir par exemple l'article de Dezan et Quinton [26]).

Un autre exemple, légèrement plus complexe, de propriété nécessitant une preuve par induction est donné par Rajopadhye dans [62]. L'auteur de cet article cherche notamment à calculer un ordonnancement pour un certain programme. Pour cela, il remplace (informellement !) toutes les équations du programme, de la forme `app[i,j,k] = app[i,k,k] * app[i,j,k-1]` par des équations de la forme `free[i,j,k] = 1[] + (free[i,k,k] max free[i,j,k-1])`. Cette transformation doit permettre de trouver un ordonnancement au plus tôt. Le programme ainsi transformé donne

```
system Free : ({ N | N>=2 } parameter)
returns (free : { i,j,k | 0<=k<=N; 0<i,j<=N } of integer);
let
    free[i,j,k] = case
            {| k=0} : 0[];
            {| k>0; i=j=k} : 1[] + free[i,j,k-1];
            {| k>0; j=k; i>k} | {| k>0; j=k; i<k} :
                    1[] + (free[i,j,k-1] max free[k,j,k]);
            {| k>0; i=k; j>k} | {| k>0; i=k; j<k} :
                    1[] + (free[i,k,k] max free[i,j,k-1]);
            {| k>0; i>k>j} | {| k>0; j>k>i} | {| k>0; j,i>k} | {| k>0; i,j<k} :
                    1[] + (free[i,j,k-1] max free[i,k,k] max free[k,j,k]);
                esac;
tel;
```

1. C'est d'ailleurs toujours de cette façon que de tels compteurs sont définis en ALPHA, puisqu'il est impossible d'utiliser une variable d'indice dans une expression de définition : `X[t]=t` n'est pas une équation légale en ALPHA.

Une dose suffisante d'intuition permet de constater qu'il existe une formulation beaucoup plus simple, et non récursive, du même calcul :

```
system Closed : ({ N | N>=2 } parameter)
returns (closed : { i,j,k | 0<=k<=N; 0<i,j<=N } of integer);
var idxk : { x |} of integer;
let
    idxk[x] = case
        {|x<0} : idxk[x+1] - 1[];
        {|x=0} : 0[];
        {|x>0} : idxk[x-1] + 1[];
    esac;
    closed[i,j,k] = case
            {| k=0} : 0[];
            {| k>0; i=j=k} : 3[]*idxk[k] - 2[];
            {| k>0; j=k; i>k} | {| k>0; j=k; i<k}
                | {| k>0; i=k; j>k} | {| k>0; i=k; j<k} : 3[]*idxk[k] - 1[];
            {| k>0; i>k>j} | {| k>0; j>k>i}
                | {| k>0; j,i>k} | {| k>0; i,j<k} : 3[]*idxk[k];
                    esac;
tel;
```

On remarquera qu'il est nécessaire d'introduire un champ de données auxiliaire *idxk* qui sert de compteur pour exprimer $idxk = k$. Ici encore, une preuve « à la main » a été nécessaire pour montrer l'équivalence des deux formulations. La définition d'un cadre formel permettrait de généraliser ce type de résultat, impossible à obtenir par réécriture.

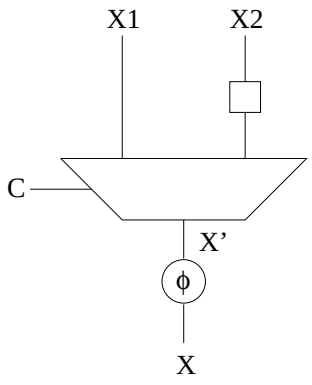
1.2 Propriétés partielles

Un autre inconvénient de l'approche interne est l'impossibilité d'abstraire les propriétés qui nous intéressent. Le raisonnement par équivalence nous oblige à considérer systématiquement toute la sémantique du programme, même si nous ne sommes intéressé que par une propriété partielle. Reprenons l'exemple de la convolution. Il est possible de prouver à la main que ce programme simple calcule effectivement une convolution. Mais nous pouvons avoir envie de savoir uniquement que, sous certaines hypothèses sur les entrées, il n'y aura pas de dépassement de capacité des variables pendant le calcul. Dans ce cas, le calcul exact de toutes les variables est inutile, et coûteux en temps. Supposons sur cet exemple que les coefficients de la convolution sont bornés par M et que les valeurs d'entrée sont bornées par K (ces deux hypothèses tiennent lieu de précondition). Je veux alors simplement montrer que les valeurs de sortie sont bornées par $4KM$ (postcondition). La preuve de cette propriété ne devrait alors pas nécessiter le passage par un calcul des valeurs exactes des sorties. Il nous faut donc définir une méthode de preuve permettant d'exprimer des propriétés plus « abstraites ».

1.3 Propriétés non exprimables en ALPHA

La dernière catégorie — et la plus intéressante — des propriétés qui apparaissent nécessaires est celle concernant des propriétés que l'on ne peut exprimer en ALPHA. Nous avons vu dans les

sections précédentes des expressions qui n'existent pas en ALPHA, comme dans le cas du compteur `X[t]=t`. De telles restrictions sont davantage dues au but initial poursuivi par la définition du langage — décrire des circuits systoliques — qu'à sa nature même. Il existe d'autres problèmes, qui seraient difficiles à résoudre sans altérer profondément le langage, au risque de perdre l'avantage de l'automatisation pour nombre de transformations. Prenons un exemple.




```
system hardware_share: ({N|N>=0} parameter;
   X1,X2: {t| t>=0;t<=N+1} of integer;
   C:{t|t>=0} of boolean)
returns (X: {t|t>=0;t<=N+1} of integer);
var
   Xprim: {t| t>=0;t<=N+1} of integer;
let
   X[t] = Xprim[t]*2[];
   Xprim[t] = if C[t] then
          case {|0<=t<=N}: X1[t];
               {|t=N+1}: 0[];
          esac else
          case {|1<=t<=N+1}: X2[t-1];
               {|t=0}: 0[];
          esac;
tel;
```

FIG. 1.1 – *Un exemple de circuit (multiplexeur)*

Le programme en figure 1.1 décrit un circuit composé d'un opérateur ϕ (une multiplication par 2 sur cet exemple de programme), et d'un multiplexeur. Suivant la valeur du contrôle booléen C, le multiplexeur transmet soit la première entrée soit la seconde sur la sortie. La deuxième entrée passe en outre par un registre qui permet de la retarder d'un cycle d'horloge. Supposons maintenant que le contrôle soit spécifié de la façon suivante :

```
C[t] = case {|t=0}: true;
            {|t>0}: not C[t-1];
       esac;
```

Intuitivement, il est clair que pour les valeurs paires de t, $C[t] = \mathit{true}$, donc $X[t] = X_1[t]$, et pour les valeurs impaires $C[t] = \mathit{false}$, donc $X[t] = X_2[t-1]$. Cependant, même cette propriété très simple ne peut être prouvée uniquement au moyen de réécritures :

- elle ne peut même pas être exprimée : `{|u>=0}:X[2u]=X1[2u]` n'est pas une expression légale en ALPHA, car l'ensemble des entiers pairs n'est pas un domaine polyédrique ;
- la preuve de cette propriété fait intervenir une induction sur un domaine non borné, ce que l'on ne peut appréhender par des réécritures.

Ces exemples divers montrent qu'il existe de nombreuses propriétés que l'on souhaiterait prouver sur des programmes ALPHA, principalement au niveau des spécifications, et que les propriétés les plus intéressantes ne peuvent être établies si l'on se cantonne à une approche interne. Nous allons donc définir un cadre externe pour la validation de programmes ALPHA, s'appuyant sur un langage plus « riche ».

Remarquons finalement que, plus le programme ALPHA a une taille importante, plus le besoin d'« abstraire » des propriétés à partir de ce programme se fait sentir. Ceci est d'autant plus vrai dans

le cas de programmes structurés par l'introduction de modules, tels qu'introduits par de Dinechin dans sa thèse [30]. Avec de tels programmes, il ne paraît pas « raisonnable » de tenir compte de *tous* les calculs effectués par *tous* les sous-programmes. Abstraire des propriétés *modulaires* est un enjeu crucial, que nous abordons dans le dernier chapitre de cette seconde partie.

Plan

Le contenu de cette seconde partie est le suivant : nous donnons tout d'abord une rapide description du langage ALPHA dans le chapitre 2. Nous définissons ensuite une sémantique opérationnelle (chapitre 3) qui sert de base à l'établissement d'un cadre logique de spécification et de preuve (chapitre 4). Le chapitre 5 donne une méthodologie de preuve s'inscrivant dans ce cadre, et le chapitre 6 donne une première piste pour envisager des preuves modulaires.

chapitre 2

Le langage ALPHA

Dans ce chapitre, nous donnons tout d'abord une rapide description du langage ALPHA tel qu'il a été défini dans [52]. Une description assez complète peut être trouvée dans [69]. Cette description du langage en donne une vision de type macroscopique (selon la terminologie introduite par Bougé [3]), où les objets de base (variables et expressions) sont des polyèdres. Nous donnons ensuite une vision de type microscopique du langage, où les variables sont déstructurées en leurs composantes élémentaires. Cette vision nous sera utile pour la définition de la sémantique opérationnelle.

Le langage ALPHA a été défini par Mauras [52]. Il est fondé sur le formalisme des *équations récurrentes* de Winograd [43], et plus précisément sur le modèle des *équations récurrentes affines*. Les programmes ALPHA opèrent sur des données définies sur des domaines polyédriques, avec des relations de dépendance affines. Domaines et dépendances sont définis statiquement.

Contrairement à certains langages voisins de type LUSTRE [17], ALPHA ne connaît pas a priori de notion de temps. Un programme ALPHA ne fait que décrire un ensemble de calculs. Les dépendances entre ces calculs imposent des restrictions sur l'ordre dans lequel ceux-ci sont exécutés, mais sans que cet ordre soit total ou explicite. Dans la plupart des cas, il peut y avoir plusieurs ordres d'exécution donnant le même résultat. Il est parfois possible d'interpréter une dimension des domaines comme étant le temps, mais ce n'est qu'une interprétation parmi d'autres. Ces aspects seront discutés plus précisément dans le chapitre suivant.

Le langage ALPHA est dit *à assignation unique* : chaque point du domaine d'une variable polyédrique ne peut avoir qu'une valeur « utile » ; cette valeur ne sera pas écrasée par un calcul ultérieur.

Nous présentons maintenant les objets et structures du langage. Dans la suite, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ et $\mathbb{B}$ dénoteront respectivement les ensembles des entiers relatifs, des rationnels, des réels et des booléens.

2.1 Présentation de ALPHA : une vision macroscopique

2.1.1 Variables

Une *variable* en ALPHA est un ensemble multidimensionnel dénombrable de valeurs, qui sont toutes de même type (entier, réel ou booléen). Plus formellement, une variable ALPHA peut être vue comme une fonction partielle de $\mathbb{Z}^n \to \mathbb{Z}$, $\mathbb{Z}^n \to \mathbb{R}$ ou $\mathbb{Z}^n \to \mathbb{B}$, où n est un entier positif. Cette fonction possède un domaine de définition donné statiquement par la déclaration de la variable, et que nous appellerons simplement *domaine* de la variable. Ce domaine est défini comme l'union de polyèdres convexes de dimension n. Plus formellement, nous donnons les définitions suivantes.

Définition 15 (Polyèdre) *Un polyèdre $\mathcal{P}$ de dimension n est un sous-ensemble de $\mathbb{Q}^n$ borné par un*

nombre fini d'hyperplans[1].

Définition 16 (Domaine) *Un domaine polyédrique $\mathcal{D}$ de dimension n est défini comme*

$$\mathcal{D} = \mathbb{Z}^n \cap \mathcal{P}$$

où $\mathcal{P}$ est une union de polyèdres de dimension n.

La définition d'une variable peut donc être formalisée de la façon suivante.

Définition 17 (Variable) *Une variable X est une application d'un domaine dans un ensemble d'un certain type.*

$$X : \mathcal{D} \mapsto \mathcal{T}$$

où $\mathcal{T} = \mathbb{Z}$ ou $\mathbb{R}$ ou $\mathbb{B}$.

Remarque : les variables scalaires sont définies sur le domaine trivial $\mathbb{Z}^0$.

Par exemple, la déclaration suivante

```
X: {i,j | 0 <= i <= j <= 2}
```

est celle d'une variable ayant un domaine triangulaire de la forme suivante

$$\begin{array}{cccc} & \xrightarrow{j} & & \\ i\downarrow & X_{0,0} & X_{0,1} & X_{0,2} \\ & & X_{1,1} & X_{1,2} \\ & & & X_{2,2} \end{array}$$

Les indices i et j donnés ici sont bien entendu muets.

Des domaines plus complexes peuvent être définis grâce aux opérateurs |, & et &$\sim$ représentant respectivement l'union, l'intersection et la différence. Le point important est que l'ensemble des unions de polyèdres convexes est stable par toutes ces opérations. Les domaines peuvent naturellement être non bornés. Par exemple,

```
{i,j,k | k=i+j}
```

représente un plan dans un espace à trois dimensions.

2.1.2 Expressions

Les expressions ALPHA sont construites à partir des variables et des constantes, combinées par les opérateurs décrits ci-dessous. Les expressions possèdent donc, comme les variables, un type et un domaine, inférés statiquement à partir de ceux de leurs constituants.

Les opérateurs utilisés pour construire les expressions sont de trois types : les opérateurs *immobiles* décrivent des opérations point-à-point, les opérateurs *spatiaux* manipulent les domaines et permettent d'exprimer des communications, et l'opérateur de *réduction* introduit ultérieurement par Le Verge [48] combine ces deux types d'opérations.

1. Rappelons qu'une équation affine définit un hyperplan.

2.1.3 Opérateurs immobiles

Ce sont la plupart des opérations arithmétiques et relations logiques scalaires usuelles, étendus composante par composante aux domaines polyédriques. Ce type d'extension est tout à fait semblable à celui déjà présenté pour le langage $\mathcal{L}$ au début de ce manuscrit. Par exemple, si l'on a les déclarations suivantes pour les variables X et Y :

```
X : {i,j | 0 <= i <= j <= 2}
Y : {i,j | 0 <= i <= 2, 0 <= j <= 2}
```

la somme $X + Y$ sera l'ensemble des valeurs

$$\begin{array}{lll} X_{0,0}+Y_{0,0} & X_{0,1}+Y_{0,1} & X_{0,2}+Y_{0,2} \\ & X_{1,1}+Y_{1,1} & X_{1,2}+Y_{1,2} \\ & & X_{2,2}+Y_{2,2} \end{array}$$

Le domaine d'une expression définie par un opérateur immobile est l'intersection des domaines de ses sous-expressions[2]. Sur l'exemple de la somme $X + Y$, le domaine de la somme est le domaine de X, qui est l'intersection des deux domaines.

2.1.4 Opérateurs spatiaux

Les opérateurs spatiaux sont de trois types.

- L'opérateur de *restriction*, noté « : », restreint une expression à un sous-domaine. Si l'expression E est définie sur le domaine $\mathcal{D}$, alors $\mathcal{D}' : E$ est définie sur $\mathcal{D} \cap \mathcal{D}'$ et possède les mêmes valeurs que E sur cette intersection. La syntaxe permettant de décrire $\mathcal{D}'$ est la même que celle utilisée pour les déclarations de variables.

- L'opérateur d'union (case) permet de rassembler des sous-expressions ayant des domaines disjoints. Les sous-expressions forment un ensemble $\{e_i\}_{1 \leq i \leq n}$ dont les domaines respectifs sont les $\{\mathcal{D}_i\}_{1 \leq i \leq n}$, tels que $\bigcap_{i=1}^{n} \mathcal{D}_i = \emptyset$, et l'expression case $e_1; \ldots; e_n$ esac est définie sur l'union $\bigcup_{i=1}^{n} \mathcal{D}_i$ et vaut e_i en un point $z \in \mathcal{D}_i$. Dans la plupart des cas, les expressions e_i utilisent l'opérateur de restriction afin d'assurer la disjonction des domaines.

- L'opérateur de *dépendance*, noté « . », permet d'exprimer une communication. Il admet comme arguments une expression e et une fonction affine f sur les indices. La valeur de l'expression $e.f$ au point z est la valeur de l'expression e au point $f(z)$. Si $\mathcal{D}$ est le domaine de e, le domaine de $e.f$ est donc $f^{-1}(\mathcal{D})$. La syntaxe des fonctions affines est du type (i,j -> 2i-1,i+j), les variables i et j étant muettes. Cette fonction pourrait être dénotée plus classiquement par $\lambda(i,j).(2i-1, i+j)$. La dimension de l'ensemble d'arrivée de la fonction n'est pas obligatoirement la même que celle de l'ensemble de départ. Par exemple, (i -> i,i+1) est également une fonction de dépendance valide.

2.1.5 Opérateur de réduction

Cet opérateur, introduit par Le Verge dans sa thèse [48], permet d'associer un opérateur point-à-point associatif et commutatif à une opération de communication de type *gather* (de plusieurs sources vers un même but). Considérons une expression e de domaine $\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{P}_i$, un opérateur

2. Cette intersection est vide si les domaines ne sont pas de même dimension.

commutatif et associatif $\oplus$ possédant un élément neutre, et une fonction affine f de $\mathcal{D}$ dans $\mathbb{Z}^p$. L'expression **reduce** $(\oplus, f, e)$ est définie sur l'union des enveloppes convexes[3] des $f(\mathcal{P}_i)$ et a pour valeur, en un point z de ce domaine, $\bigoplus_{\{z' \in \mathcal{D} | f(z')=z\}} e(z')$. En général, la fonction f est une projection. Par exemple, le calcul de la somme

$$\sum_{j=1}^{N} a_{ij} b_j$$

s'exprimera en ALPHA

```
reduce(+,(i,j->i),a*b.(i,j->j))
```

Ici, f est une projection sur i.

Aux indices z tels que l'ensemble $\{z' \in \mathcal{D} \mid f(z') = z\}$ est vide, l'expression a pour valeur l'élément neutre de l'opérateur $\oplus$.

2.1.6 Programmes

Un programme ALPHA est appelé *système*. Il comprend un en-tête comprenant des déclarations et un ensemble d'équations délimité par les mots-clefs **let** et **tel**.

L'en-tête permet de distinguer trois classes de variables : les variables d'entrée, les variables de sortie (placées après le mot-clef **returns**), et les variables locales (placées après le mot-clef **var**). L'en-tête permet donc de définir les noms et les domaines de ces variables. L'en-tête définit aussi des paramètres de taille utilisés dans les déclarations de domaines.

Une équation associe une variable et une expression. Il s'agit d'une équation de définition et non d'une affectation. On la note

$$X = E$$

Comme exemple de programme, nous donnons deux versions d'un système décrivant un filtre de convolution (figures 2.1 et 2.2). Les variables a et x sont des variables d'entrée, y est une variable de sortie, et Y une variable locale. Le système est paramétré par N.

2.1.7 Notation tableau

L'écriture et surtout la lecture des expressions contenant des fonctions de communication n'est pas toujours aisée. Wilde [69] a donc introduit une notation plus intuitive, dite notation tableau. Ainsi, au lieu d'écrire

```
Y = A * X.(i,j ->i+j)
```

on écrira

```
Y[i,j] = A[i,j] * X[i+j]
```

Le passage de la notation initiale à la notation tableau — et réciproquement — est uniquement une manipulation syntaxique. L'exemple du programme de convolution en notation tableau est donné en figure 2.3.

Dans le cas de composition de fonctions de dépendance (dépendances imbriquées), d'opérateurs de restriction imbriqués ou d'opérateurs de réduction, cette notation n'est pas utilisable et il faut revenir à la notation initiale.

3. Remarquons qu'en général, $f(\mathcal{P}_i)$ n'est pas un domaine polyédrique. Par exemple, l'image de tout domaine non vide et non réduit à un point par `(i ->2*i)` n'est pas convexe.

```
system convolution( N : { N | N>=0 } parameter;
        a : { j | 1<=j<=N } of integer;
        x : { i | i>=1 } of integer )
    returns ( y : { i | i>=N } of integer);
    var
        Y : { i,j | 0<=j<=N ; i>=N } of integer;
    let
        Y = reduce(+,(i,j->i),x.(i,j->i-j+1)*a.(i,j->j));
        y = Y.(i->i,N);
    tel
```

FIG. 2.1 – *Programme* ALPHA *décrivant un filtre de convolution et utilisant une réduction.*

```
system convolution( N : { N | N>=0 } parameter;
        a : { j | 1<=j<=N } of integer;
        x : { i | i>=1 } of integer )
    returns ( y : { i | i>=N } of integer);
    var
        Y : { i,j | 0<=j<=N ; i>=N } of integer;
    let
        Y = case
            { i,j| j=0 } : 0.(i,j->);
            { i,j| 1<=j<=N } : Y.(i,j->i,j-1) + a.(i,j->j)*x.(i,j->i-j+1);
            esac;
        y = Y.(i->i,N);
    tel
```

FIG. 2.2 – *Programme* ALPHA *décrivant un filtre de convolution.*

```
system convolution( N : { N | N>=0 } parameter;
        a : { j | 1<=j<=N } of integer;
        x : { i | i>=1 } of integer )
    returns ( y : { i | i>=N } of integer);
    var
        Y : { i,j | 0<=j<=N ; i>=N } of integer;
    let
        Y[i,j] = case
            { | j=0 } : 0[];
            { | 1<=j<=N } : Y[i,j-1] + a[j]*x[i-j+1];
            esac;
        y[i] = Y[i,N];
    tel
```

FIG. 2.3 – *Le programme de convolution en notation tableau.*

2.2 Une vision microscopique

Nous avons donné jusqu'à maintenant une vision du langage ALPHA qui considère les polyèdres comme les entités de base, et qui opère par manipulations globales sur ces polyèdres. Nous allons compléter cette vision de type macroscopique par une autre vision, qui se place cette fois au niveau des *instances de variables polyédriques*. C'est cette vision, dite de type microscopique, qui nous servira pour la définition d'une sémantique opérationnelle.

2.2.1 Déstructuration des variables

Dans la définition du langage ALPHA, les variables sont des ensembles de composantes définies sur un domaine polyédrique. Nous utiliserons la terminologie suivante.

On appelle variable une « variable » polyédrique ALPHA prise dans le sens classique du terme.
On appelle composante un point (une instance) d'un champ de données.

Par exemple, le texte suivant

```
var Y : { i,j | 0<=j<=N ; i>=N } of integer;
```

définit une variable Y et un ensemble de composantes $Y[i,j]$.

À partir de ce point, nous n'allons plus raisonner directement sur des polyèdres, mais sur des ensembles de composantes, qui sont donc « déstructurés ».

2.2.2 Ensembles d'entrée et de sortie

Nous distinguons deux ensembles particuliers de composantes :

- les composantes de sortie, qui correspondent aux instances des variables déclarées en sortie d'un système ;

- les composantes d'entrée, qui correspondent d'une part aux instances des variables déclarées en entrée d'un système, et d'autre part aux composantes dont l'expression de définition est « syntaxiquement » constante, c'est-à-dire ne fait pas intervenir de composante. Ces deux types de composantes vont en effet présenter le même comportement.

Les autres composantes seront dites *locales*. L'ensemble des composantes locales est donc un sous-ensemble de celui des instances de variables déclarées comme locales par la section `var`.

On remarquera que ces ensembles de composantes peuvent être calculés par une simple analyse syntaxique d'un programme ALPHA.

2.2.3 Équations et programmes

Une équation associe une expression de définition à une composante. Elle est notée

$$x = e(y_1, \ldots, y_n) \quad \text{ou} \quad x = e(\vec{y})$$

Ces équations peuvent être directement dérivées du texte du programme ALPHA.

Un programme est alors simplement vu comme la donnée de

- un ensemble de composantes d'entrée In ;
- un ensemble de composantes de sortie Out ;
- un ensemble d'équations $x = e(\vec{y})$, d'au plus une équation pour chaque composante x.

2.2.4 Graphe de dépendances

Le graphe de dépendance d'un programme ALPHA permet d'abstraire celui-ci en éliminant les calculs pour ne garder que les directions des mouvements de données entre composantes. Les nœuds du graphe sont les composantes, et il y a un arc entre deux composantes si l'une dépend directement de l'autre : on dit qu'une composante x dépend directement d'une composante y s'il existe une équation de définition $x = e(\ldots, y, \ldots)$.

Il existe pour les systèmes d'équations récurrentes uniformes ou affines d'autres notions de graphes de dépendance tirant profit de la régularité de celles-ci, mais elles ne nous seront pas utiles ici.

Une notation classique pour traduire le fait qu'une composante x dépend d'une composante y consiste à écrire [4]

$$x \longrightarrow y.$$

Nous utiliserons une autre notation, préférant conserver la flèche pour d'autres usages. Nous nous intéresserons en effet aux graphes de dépendance sans cycles, et plutôt à l'ordre partiel induit dans ce cas par le graphe de dépendance. Nous noterons alors $\preceq$ la fermeture réflexive et transitive de la relation de dépendance directe :

$$x \preceq y$$

s'il existe un chemin (éventuellement de longueur nulle) de y à x dans le graphe de dépendance, c'est-à-dire s'il existe une suite $(x_0, \ldots, x_n)$ telle que $x_0 = x$, $x_n = y$, et pour tout i, $1 \leq i \leq n$, $x_i = e(\ldots, x_{i-1}, \ldots)$ est l'équation de définition de x_i. Un tel chemin est appelé *chemin de dépendance*.

4. Attention au sens de la flèche !

2.2.5 Programmes bien formés

Nous avons donné une définition très générale de la notion de programme, qui dit que tout ensemble d'équations de définition contenant au plus une équation par composante est un programme. Nos aurons besoin d'une notion plus restreinte, qui correspond en fait aux programmes « utiles » et « exécutables »[5] :

Un programme *bien formé* est un programme respectant les deux conditions suivantes :

- il y a une et une seule équation de définition pour chaque composante qui n'est pas une composante d'entrée ;
- le graphe de dépendance est sans cycle[6].

Dans la suite, nous ne considérerons que des programmes bien formés.

5. Cette notion sera précisée au chapitre suivant.
6. Contrairement au point précédent, celui-ci n'est malheureusement pas décidable par analyse syntaxique.

chapitre 3

Deux sémantiques opérationnelles

Comme nous l'avons vu dans le chapitre précédent, la définition d'un programme ALPHA ne spécifie pas l'ordre d'évaluation des expressions. Un même programme peut avoir plusieurs *chemins d'évaluation*, dont la réunion forme un graphe d'exécution (voir par exemple le rapport de DEA de Nédelka, [54]). Par ailleurs, même si parfois plusieurs instances d'une même variable peuvent être évaluées en parallèle, les évaluations ne se font pas de façon monolithique pour ces variables. Chaque transition dans le graphe d'exécution correspond à l'évaluation d'un seul point d'une variable, c'est-à-dire à une composante dans notre vision microcopique. C'est ici en effet que cette vision s'avère utile : plutôt que de raisonner sur des variables structurées, nous allons travailler avec des transitions au sein d'ensembles homogènes de composantes.

Le mémoire de DEA de Nédelka [54] donne une première idée de ce que serait une sémantique opérationnelle pour ALPHA. Il propose en particulier deux approches, qui diffèrent selon la façon dont le graphe d'exécution est généré. La première, de type « flot de données », consiste à évaluer une expression dès que les composantes libres sont définies. La seconde, « à la demande », n'évalue que des expressions utiles au calcul d'une composante de sortie. Après avoir précisé les notations que nous emploierons, nous reprenons ici ces deux approches et établissons leur équivalence.

3.1 Notations

Valeur indéfinie Pour tout ce qui concerne la sémantique opérationnelle, nos ne tenons pas compte du type des composantes. Nous considérons en revanche un élément particulier de l'ensemble des valeurs, que nous notons $\perp$, et qui dénote la non-définition. L'ensemble des valeurs sera donc $\mathcal{V} \cup \perp$, où $\mathcal{V}$ représente l'ensemble des valeurs définies.

États Un état (ou environnement) σ est une application de l'ensemble des composantes dans l'ensemble des valeurs. Le support de σ est l'ensemble des composantes ayant une valeur définie dans σ. On le note $supp(\sigma)$:

$$supp(\sigma) = \{x \mid \sigma(x) \neq \perp\}$$

Les états sont généralisés aux expressions de façon usuelle. Nous ne détaillons pas ici la fonction de calcul des expressions, tout à fait classique.

États initiaux et finaux Pour définir une sémantique opérationnelle, nous allons raisonner à partir de transitions : une transition part d'un état σ et donne un état σ' après évaluation d'une composante. Il nous faut alors distinguer deux notions particulières d'états : les états initiaux et les

états finaux. La notion d'état initial n'est pas très difficile à préciser : on dit qu'un état est initial si toutes les composantes d'entrée — et seulement celles-ci — ont une valeur définie dans cet état. Rappelons que l'ensemble des composantes d'entrée comprend les composantes dont l'expression de définition est constante. Si l'on se place du point de vue « description de circuit », cela revient à considérer que ces composantes sont des registres que l'on charge avec une valeur constante à l'initialisation du circuit. Nous notons *Init* l'ensemble des états initiaux :

$$\sigma \in Init \quad \Leftrightarrow supp(\sigma) = In$$

La notion d'état final est plus difficile à mettre en œuvre. En effet, il serait naturel de dire qu'un état final est atteint lorsque toutes les composantes sont définies. Or l'ensemble des composantes peut être infini. De plus, seules les composantes de sortie nous intéressent, et de par la nature flot de données du langage, il se peut que toutes les composantes de sortie soient valuées alors qu'il reste des transitions possibles. Un état final est donc défini par rapport à un sous-ensemble fini des composantes de sortie : considérons V un sous-ensemble fini de Out. Alors σ est un état final par rapport à V si et seulement si toutes les composantes de V sont valuées dans σ.

3.2 L'approche « flot de données »

Dans cette approche, nous considérons un programme comme un « producteur » de valeurs de sortie : l'évaluation part des composantes d'entrée, qui ont une valeur définie dans l'état initial, et calcule les valeurs des autres composantes dès que les expressions de définition de celles-ci peuvent l'être. La figure 3.1) illustre ce mécanisme : les disques noirs représentent les composantes ayant une valeur, et les cercles celles qui sont encore indéfinies. Les flèches représentent (en partie) les dépendances entre composantes. La seule chose que nous ayons besoin de savoir est le fait que les

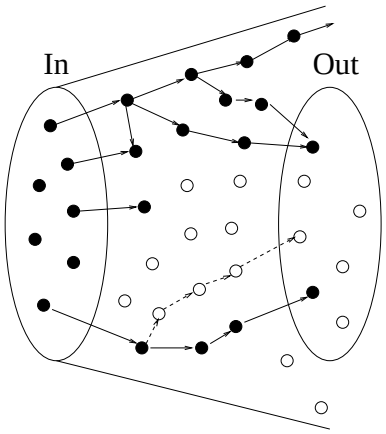


FIG. 3.1 – *L'approche « flot de données »*

composantes libres d'une expression de définition sont toutes définies. Le calcul ne s'appliquera que si la composante considérée n'est pas déjà définie. Plus formellement, nous donnons la règle suivante, où $x = e(y_1, \ldots, y_n)$ est l'expression de définition de x.

Règle 7 (Flot de données)

$$\frac{(\forall i \in \{1..n\} : \sigma(y_i) \neq \bot) \wedge \sigma(x) = \bot}{\sigma \xrightarrow{x} \sigma[x \leftarrow \sigma(e(y_1, \ldots, y_n))]}$$

Une suite d'états $(\sigma_i)_{i\in\{0,..,n\}}$ *telle que* $\forall i \in \{1,..,n\}, \sigma_{i-1} \xrightarrow{x_i} \sigma_i$ *est appelée une* suite de transitions.

La règle donnée ici est non-déterministe : à chaque étape, on choisit parmi une des composantes « possibles » celle qui sera valuée, sans autre restriction que celles imposées par le graphe de dépendances.

3.3 L'approche « à la demande »

L'approche précédente permet de valuer toutes les composantes, dès lors que leur expression de définition est évaluable. Cette approche permet donc de calculer des composantes qui ne seront pas nécessaires, dans le sens où elles ne sont pas utiles au calcul des sorties. Par exemple, si l'on a le programme suivant

```
system S ( a : { |} of integer )
    returns (X : {t | t>= 0} of integer ;
var
    X' : {t | t>= 0} of integer ;
let
    X[t] = X'[t];
    X'[t] = case
        { | t=0 } : a[];
        { | t >0 } : X'[t-1];
    esac;
tel;
```

Dans ce programme, les composantes de sortie sont les $X[t]$ et les composantes locales les $X'[t]$. La règle d'évaluation de type flot de données peut conduire à évaluer successivement les composantes locales, sans jamais évaluer les composantes de sortie. Si de plus nous remplaçons l'équation de définition de X par

```
X[t] = X'[0]
```

et si nous considérons la même suite d'évaluations, nous aurons une suite infinie d'évaluations qui sont « superflues », dans le sens où elles ne sont utiles à aucun calcul de composante de sortie.

Nous introduisons donc une seconde approche, qui permet de restreindre les évaluations à celles qui sont « nécessaires », dans le sens que nous précisons ci-dessous. Nous avons vu lors de la définition des états finaux que ceux-ci étaient liés à un sous-ensemble fini de composantes de sortie. De la même façon, nous allons nous intéresser à des ensembles de composantes nécessaires au calcul d'un sous-ensemble de composantes de sortie. Nous donnons ici la définition formelle d'un tel ensemble.

Définition 18 (Nécessaire) *Soit V un sous-ensemble de Out. Les composantes* nécessaires à V *sont les composantes apparaissant sur un chemin de dépendance entre In et V.*

$$\mathit{Nécessaire}(V) = \{x \mid \exists z \in In, \exists t \in V,\ z \preceq x \preceq t\}$$

Notons que dans cette définition V n'est pas obligatoirement fini. Nous mentionnerons cette restriction chaque fois qu'il y aura lieu. Par convention, l'ensemble $\mathit{Nécessaire}(\mathit{Out})$ sera simplement noté *Nécessaire*.

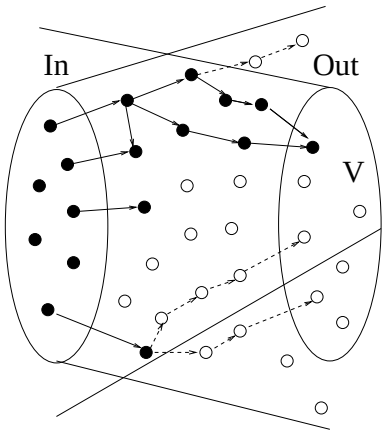


FIG. 3.2 – *L'approche « à la demande »*

La nouvelle règle de transition diffère alors de la précédente par le fait qu'elle vérifie que la composante à valuer est nécessaire(cf. figure 3.2). Les transitions sont donc étiquetées par le sous-ensemble V considéré.

Règle 8 (À la demande)

$$\frac{(\forall i \in \{1..n\} : \sigma(y_i) \neq \bot) \wedge \sigma(x) = \bot \wedge x \in \mathit{Nécessaire}(V)}{\sigma \underset{V}{\overset{x}{\Longrightarrow}} \sigma[x \leftarrow \sigma(e(y_1, \ldots, y_n))]}$$

Dans le cas où $V = Out$, nous écrirons simplement $\sigma \overset{x}{\Longrightarrow} \sigma'$.

Sur l'exemple que nous avons donné ci-dessus, et si l'on utilise maintenant la seconde règle, avec comme sous-ensemble V l'ensemble $\{1..N\}$, les composantes $X'[t]$ ne seront plus évaluées pour $t > N$.

3.4 Équivalence entre les deux approches

L'approche de type « à la demande » est une restriction de l'approche flot de données, qui est la plus générale dans le sens où elle donne le graphe de toutes les exécutions possibles. Nous aimerions montrer que la restriction que nous avons opérée pour la deuxième approche n'est pas trop forte, et donc que les deux approches sont en un certain sens équivalentes. Elles ne peuvent pas l'être dans un sens strict, puisque nous savons déjà que toutes les exécutions générées par la première ne peuvent l'être par la seconde. Ce qui nous intéresse ici, ce sont les valeurs des composantes de sortie. L'équivalence est donc définie par rapport à un ensemble de composantes de sortie. Nous donnons d'abord une définition d'équivalence modulo un ensemble de composantes quelconque.

Définition 19 (Équivalence modulo un ensemble de composantes) *Soit V un ensemble de composantes, et σ,σ' deux états. On dit que σ et σ' sont* équivalents modulo V, *$\sigma \underset{V}{\sim} \sigma'$ si et seulement si*

$$V \subseteq \mathit{supp}(\sigma) \qquad V \subseteq \mathit{supp}(\sigma') \qquad \forall x \in V,\ \sigma(x) = \sigma'(x).$$

Nous voulons maintenant montrer que les états finaux obtenus par une suite de transitions utilisant la seconde règle sont équivalents à ceux obtenus grâce à la première règle. Plus formellement, nous allons montrer le théorème suivant, dont la réciproque est trivialement vraie.

Théorème 10 (Équivalence) *Soit V un sous-ensemble fini de Out, et σ_0 un état initial. S'il existe une suite de transitions*

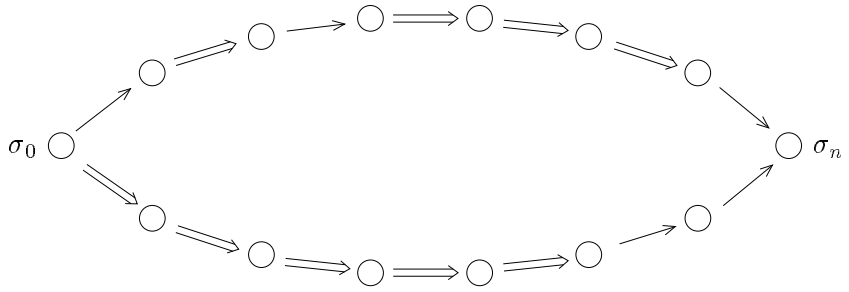
$$\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-1}} \sigma_n$$

avec $V \subseteq supp(\sigma_n)$ (c'est-à-dire que σ_n est un état final par rapport à V), alors il existe une suite

$$\sigma_0 \overset{x'_0}{\underset{V}{\Longrightarrow}} \sigma'_1 \overset{x'_1}{\underset{V}{\Longrightarrow}} \ldots \overset{x'_{m-1}}{\underset{V}{\Longrightarrow}} \sigma'_m$$

où les x'_i sont des éléments distincts de $\{x_0, \ldots, x_n\}$ et avec $\sigma_n \underset{V}{\sim} \sigma'_m$.

Nous allons tout d'abord montrer qu'il est possible de *réarranger* une suite de transitions flot de données, afin d'évaluer dans un premier temps toutes les composantes nécessaires, puis dans un second temps les composantes non nécessaires.



Un tel réarrangement n'est pas juste une permutation des états qui composent la suite de transitions. Comme l'ordre d'évaluation des composantes n'est pas le même, les états ne sont pas « enrichis » par les nouveaux calculs de la même façon. Seuls les états finaux des deux suites seront les mêmes, après évaluation de toutes les composantes. Pour montrer que l'état final obtenu après évaluation des composantes nécessaires est équivalent à l'état final de la suite de transitions initiale, il nous faut prouver que l'incrément est le même, c'est-à-dire que l'évaluation de l'expression de définition d'une composante donnera la même valeur dans les deux suites de transitions. Nous montrons donc tout d'abord le lemme suivant, qui exprime le fait que la valeur d'une composante ne change pas à partir du moment où celle-ci a été calculée (et est donc devenue différente de $\perp$). La clef ici provient du fait que le langage est à assignation unique. Un résultat similaire a été énoncé et discuté informellement par Nédelka [54]. Nous en donnons ici une formulation et une preuve précises.

Lemme 9 (Conservation de la valeur) *Soit S un programme, $\sigma_0 \to \sigma_1 \to \ldots \to \sigma_k$ une suite de transitions pour ce programme, x une composante de S. Si $\sigma_0(x) \neq \perp$, alors pour tout $0 \leq j \leq k$, nous avons $\sigma_j(x) = \sigma_0(x)$.*

Preuve. *Nous montrons par récurrence sur j dans $\{0, .., k\}$ que nous avons $\sigma_j(x) = \sigma_0(x)$.*

Ceci est trivialement vrai pour $j = 0$.

Supposons que cette propriété ait été établie pour j, avec $j < k$. Par hypothèse de récurrence, nous avons $\sigma_j(x) = \sigma_0(x)$, donc $\sigma_j(x) \neq \perp$. La transition $\sigma_j \to \sigma_{j+1}$ correspond donc à

l'évaluation d'une composante x_j, avec $x_j \neq x$: $\sigma_{j+1} = \sigma_j[x_j \leftarrow \sigma_j(e(\vec{y_j}))]$. Comme l'évaluation des expressions ALPHA *ne produit pas d'effets de bord, nous avons $\sigma_{j+1}(x) = \sigma_0(x)$. La propriété est donc établie pour $j+1$.* ❑

Nous pouvons maintenant prouver la proposition suivante, qui est la formalisation de la propriété de réarrangement.

Proposition 11 (Réarrangement) *Soit V un sous-ensemble fini de Out, et σ_0 un état initial. S'il existe une suite de transitions*

$$\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-1}} \sigma_n$$

avec $V \subseteq supp(\sigma_n)$, alors il existe une suite de transitions

$$\sigma_0 = \sigma'_0 \overset{x'_0}{\underset{V}{\Longrightarrow}} \sigma'_1 \overset{x'_1}{\underset{V}{\Longrightarrow}} \ldots \overset{x'_{k-1}}{\underset{V}{\Longrightarrow}} \sigma'_k \xrightarrow{x'_k} \sigma'_{k+1} \xrightarrow{x'_{k+1}} \ldots \xrightarrow{x'_{n-1}} \sigma'_n = \sigma_n$$

telle que

$$\begin{cases} (x'_0, \ldots, x'_{n-1}) \text{ est une permutation de } (x_0, \ldots, x_{n-1}) \\ \forall j \in \{k, .., n-1\},\ x'_j \notin N\acute{e}cessaire(V) \\ \sigma'_k \underset{V}{\sim} \sigma_n \end{cases}$$

Nous donnons ici la preuve de cette proposition, technique mais assez caractéristique de la méthode à appliquer.

Preuve. *La preuve est faite par récurrence sur la longueur n de la suite de transitions. Le résultat est trivial pour une suite de longueur 1. Supposons que nous l'ayons établi pour toute suite de longueur $n-1$, et considérons une suite $\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-1}} \sigma_n$ vérifiant les hypothèses. Nous distinguons entre deux cas, suivant le statut de x_{n-1}.*

- *Si $x_{n-1} \notin N\acute{e}cessaire(V)$. Nous appliquons l'hypothèse de récurrence à $\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-2}} \sigma_{n-1}$. Nous obtenons alors une suite*

  $$\sigma_0 = \sigma'_0 \overset{x'_0}{\underset{V}{\Longrightarrow}} \sigma'_1 \overset{x'_1}{\underset{V}{\Longrightarrow}} \ldots \overset{x'_{k-1}}{\underset{V}{\Longrightarrow}} \sigma'_k \xrightarrow{x'_k} \sigma'_{k+1} \xrightarrow{x'_{k+1}} \ldots \xrightarrow{x'_{n-2}} \sigma'_{n-1} = \sigma_{n-1}$$

  où $(x'_0, \ldots, x'_{n-2})$ est une permutation de $(x_0, \ldots, x_{n-2})$ et $\sigma'_k \underset{V}{\sim} \sigma_{n-1}$. Comme $x_{n-1} \notin N\acute{e}cessaire(V)$ et $V \subseteq N\acute{e}cessaire(V)$, nous avons $x_{n-1} \notin V$, donc $\sigma'_k \underset{V}{\sim} \sigma_n$. En ajoutant la transition $\sigma_{n-1} \xrightarrow{x_{n-1}} \sigma_n$ à la suite de transitions précédente, nous obtenons la suite désirée.
- *Si $x_{n-1} \in N\acute{e}cessaire(V)$. Nous appliquons d'abord l'hypothèse de récurrence à $\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-2}} \sigma_{n-1}$. Nous obtenons alors la suite*

  $$\sigma_0 = \sigma'_0 \overset{x'_0}{\underset{V}{\Longrightarrow}} \sigma'_1 \overset{x'_1}{\underset{V}{\Longrightarrow}} \ldots \overset{x'_{k-1}}{\underset{V}{\Longrightarrow}} \sigma'_k \xrightarrow{x'_k} \sigma'_{k+1} \xrightarrow{x'_{k+1}} \ldots \xrightarrow{x'_{n-2}} \sigma'_{n-1} = \sigma_{n-1}$$

  où $(x'_0, \ldots, x'_{n-2})$ est une permutation de $(x_0, \ldots, x_{n-2})$ et $\sigma'_k \underset{V}{\sim} \sigma_{n-1}$. Nous avons alors deux cas à considérer.

  - *Si $k = n-1$. En prenant $\sigma'_0 \overset{x'_0}{\underset{V}{\Longrightarrow}} \sigma'_1 \overset{x'_1}{\underset{V}{\Longrightarrow}} \ldots \overset{x'_{n-2}}{\underset{V}{\Longrightarrow}} \sigma'_{n-1} \overset{x_{n-1}}{\underset{V}{\Longrightarrow}} \sigma_n$, nous avons la suite désirée.*

– *Si $k < n-1$. Nous avons $x_{n-2} \notin$ Nécessaire(V). Montrons que nous n'avons pas $x_{n-2} \preceq x_{n-1}$. Dans la suite de transitions initiale, x_{n-2} a été valuée dans une séquence partant d'un état initial. Nous savons donc que x_{n-2} n'est pas dans In. De plus, l'ensemble $\{y \mid y \preceq x_{n-2}\}$ est fini, sinon x_{n-2} n'aurait pas été valué. Considérons un élément minimal pour $\preceq$ dans cet ensemble non-vide. Nous affirmons que cet élément est une composante de In. Dans le cas contraire, cette composante dépendrait d'une autre composante, puisque le programme est bien formé, et elle ne pourrait pas être minimale. Nous disposons donc d'une composante x dans In telle que $x \preceq x_{n-2}$. Si nous avions $x_{n-2} \preceq x_{n-1}$, nous aurions $x \preceq x_{n-2} \preceq x_{n-1}$. Comme $x_{n-1} \in$ Nécessaire(V), nous aurions aussi $x_{n-2} \in$ Nécessaire(V). C'est impossible, et nous concluons que $x_{n-2} \not\preceq x_{n-1}$.*
Nous pouvons maintenant « permuter » x_{n-1} et x_{n-2}. Nous savons que, dans σ'_{n-1}, l'équation de définition de x_{n-1} est évaluable, et que $\sigma'_{n-1}(x_{n-1}) = \bot$. Comme $x_{n-2} \not\preceq x_{n-1}$, ceci est également valable dans σ'_{n-2}. Nous pouvons donc construire la suite de transitions suivante :

$$\sigma'_0 \underset{V}{\overset{x'_0}{\Longrightarrow}} \sigma'_1 \underset{V}{\overset{x'_1}{\Longrightarrow}} \ldots \underset{V}{\overset{x'_{k-1}}{\Longrightarrow}} \sigma'_k \overset{x'_k}{\longrightarrow} \sigma'_{k+1} \overset{x'_{k+1}}{\longrightarrow} \ldots \sigma'_{n-2} \underset{V}{\overset{x'_{n-1}}{\Longrightarrow}} \sigma''_{n-1} \overset{x'_{n-2}}{\longrightarrow} \sigma''_n.$$

De plus, le lemme 9 nous garantit que

$$\sigma''_n = \sigma_n.$$

Nous sommes ramenés au premier cas, où nous avions $x_{n-1} \notin$ Nécessaire(V). Nous pouvons appliquer une fois de plus l'hypothèse de récurrence afin d'obtenir le résultat voulu.

Ceci termine la preuve de la proposition 11. ❑

Le théorème 10 est une conséquence directe de la proposition que nous venons de montrer.

3.5 Conséquences : déterminisme

Les propriétés que nous venons de montrer permettent d'établir immédiatement les propositions suivantes. Nous avons d'abord (après deux résultats préliminaires) une propriété de confluence de la sémantique opérationnelle : il est toujours possible de compléter deux suites de transitions partant d'un même état initial pour arriver à un même état final. La conséquence directe de cette propriété de confluence est la propriété de déterminisme : si deux suites de transitions partent d'un même état initial et aboutissent à des états finaux de même support, alors ces états finaux sont identiques.

Proposition 12 *Soit $\sigma_0 \longrightarrow^* \sigma$ une suite de transitions, et soit x une composante de Out. Si $\sigma(x) \neq \bot$, alors $\forall y \in$ Nécessaire$(\{x\}) : \sigma(y) \neq \bot$.*

Proposition 13 *Soit V un sous-ensemble fini de Out. Soit σ_0 un état initial et $\sigma_0 \underset{V}{\Longrightarrow}^* \sigma$ une suite de transitions telle que $supp(\sigma) = V$. Alors il n'existe pas de transition $\sigma \underset{V}{\Longrightarrow}$.*

Proposition 14 (Confluence) *Soit V un sous-ensemble fini de Out. Si $\sigma_0 \underset{V}{\Longrightarrow}^* \sigma_1$ et $\sigma_0 \underset{V}{\Longrightarrow}^* \sigma_2$, alors il existe un état σ' tel que $\sigma_1 \underset{V}{\Longrightarrow}^* \sigma'$ et $\sigma_2 \underset{V}{\Longrightarrow}^* \sigma'$.*

Proposition 15 (Déterminisme) *Soit V un sous-ensemble fini de Out. Si*

$$\sigma_0 \underset{V}{\Longrightarrow}^* \sigma_1$$
$$\sigma_0 \underset{V}{\Longrightarrow}^* \sigma_2$$
$$supp(\sigma_1) = supp(\sigma_2) = V,$$

alors

$$\sigma_1 = \sigma_2$$

Les résultats mentionnés dans ce chapitre sur la sémantique opérationnelle mettent en lumière le fait suivant : le non-déterminisme sur l'ordre d'évaluation des composantes n'implique pas un non-déterminisme « fonctionnel », c'est-à-dire au niveau des valeurs finales des variables. Cette conséquence directe de la propriété d'assignation unique nous a permis d'une part de montrer qu'il est possible de réordonner les transitions. Elle est d'autre part indispensable pour prouver la correction de tout schéma d'implantation d'un traducteur du langage ALPHA vers un langage simulant une exécution (C par exemple).

Il nous faut ici remarquer que dans le cas général le calcul de l'ensemble *Nécessaire* est un problème indécidable. Cet ensemble est utile pour prouver l'équivalence d'une implémentation par ordonnancement (qui est du type « flot de données ») et d'une implémentation à la demande. Il ne peut en revanche être utilisé dans un but pratique.

Nous avons utilisé ici, pour modéliser les calculs effectués par un programme ALPHA, une approche de type opérationnel. Ceci était rendu nécessaire par le fait que nous voulons pour définir ensuite notre cadre logique raisonner sur des états finaux *atteignables en un temps fini*. Mauras dans sa thèse [52] utilise une approche dénotationnelle, en définissant une sémantique par point fixe. Nous n'avons pas montré les liens entre ces deux approches. Il serait par exemple possible de montrer que toute composante valuée par la sémantique dénotationnelle de Mauras peut également être valuée par une suite de transitions de la sémantique opérationnelle flot de données, et réciproquement. Nous ne traiterons pas ici ce point de façon formelle.

chapitre 4

Un cadre logique

Nous disposons avec la sémantique opérationnelle d'une description de l'exécution d'un programme ALPHA. Celle-ci nous permet de spécifier la façon dont un programme transforme un état initial en un état final. Notre but est de prouver des propriétés sur ces états, donc d'adopter une vision plus abstraite que celle fournie par la sémantique opérationnelle. Nous avons choisi une approche externe, s'appuyant sur un langage de spécification des propriétés différent du langage de programmation lui-même. La démarche, dans ses grandes lignes, s'apparente donc fortement à celle suivie pour le langage $\mathcal{L}$:

- définir un langage d'assertion, qui nous permettra d'exprimer les propriétés à prouver ;
- définir une notion de spécification, ainsi qu'une notion de validité des spécifications ;
- définir une notion de prouvabilité d'une spécification, et prouver son adéquation avec la notion de validité.

La notion de validité reposera sur la description du comportement des programmes, c'est-à-dire sur la sémantique opérationnelle. Nous verrons que la notion de prouvabilité est ici beaucoup plus simple que dans le cas d'un langage impératif, grâce notamment à la nature « assignation unique » du langage.

4.1 Langage d'assertion

Comme nous l'avons vu dans le chapitre 1, nous souhaitons disposer d'un langage permettant d'exprimer des propriétés *fonctionnelles*, c'est-à-dire ne dépendant pas de l'ordre d'évaluation des composantes. Le langage que nous définissons ici nous permet donc simplement d'exprimer des propriétés sur les valeurs des composantes.

Comme dans le cas de la logique de Hoare, le langage de départ est la logique du premier ordre, que nous adaptons aux caractéristiques du langage de programmation. Deux points particuliers sont à prendre en compte : l'existence de variables indexées sur des polyèdres, et l'assignation unique des composantes, qui sont de ce fait indéfinies au « début » d'un calcul. Le langage que nous allons utiliser doit donc permettre de traiter la valeur indéfinie. Une solution consisterait à « étendre » les prédicats usuels de la logique du premier ordre, et en particulier le prédicat d'égalité qui doit renvoyer *true* pour la formule $\perp = \perp$. Cette solution, que nous avons adoptée dans [9], est peu satisfaisante : comment par exemple étendre le prédicat $\leq$? Nous allons ici rester dans la logique du premier ordre en ce qui concerne les expressions arithmétiques, et introduire un nouveau prédicat

pour prendre en charge les spécificités que nous devons traiter. Nous introduisons donc le prédicat suivant.

$$\Delta(X, \vec{\imath}, \alpha)$$

où X est une variable ALPHA, $\vec{\imath}$ un n-uplet d'expressions d'indice s'évaluant en un vecteur d'indice appartenant au domaine de cette variable, et α une variable de la logique du premier ordre, signifie intuitivement « $X[\vec{\imath}]$ est définie et a la valeur de α ».

Outre ce prédicat, le langage d'assertion comprend

- des expressions d'indice ; une expression d'indice est une constante entière, une variable d'indice, ou la combinaison d'expressions d'indice par les opérateurs arithmétiques usuels ;
- des variables logiques (arithmétiques ou booléennes) ;
- des expressions arithmétiques ; une expression arithmétique est une valeur constante, une variable logique arithmétique, ou la combinaison d'expressions arithmétiques par les opérateurs arithmétiques usuels ;
- des expressions booléennes; une expression booléenne est une constante booléenne, une variable logique booléenne, ou la combinaison d'expressions arithmétiques par les opérateurs de relation usuels (=,<,etc.).

Une formule du langage d'assertion est la combinaison d'expressions booléennes par les connecteurs booléens usuels ($\wedge$,$\vee$,$\neq$,$\Rightarrow$), le prédicat Δ appliqué à une variable ALPHA, un n-uplet d'expressions d'indice de même dimension que la variable et une variable logique, ou une formule quantifiée. La quantification, universelle ou existentielle, s'applique soit aux variables d'indice, soit aux variables logiques. Par exemple,

$$\forall t \geq 0 : \forall p \geq 0 : \exists \alpha : \Delta(X, (t, p), \alpha) \wedge \alpha > 0$$

signifie que X est positif sur le domaine $\mathbb{N} \times \mathbb{N}$;

$$\forall t \geq 0 : \exists \alpha : \exists \beta : \Delta(X, 2 * t, \alpha) \wedge \alpha = \mathit{true} \wedge \Delta(X, 2 * t + 1, \beta) \wedge \beta = \mathit{false}$$

signifie que $X[t]$ vaut *true* pour les valeurs paires de t, et *false* pour les valeurs impaires.

Nous ne manipulerons que des formules *closes*. Nous supposons donc que toutes les formules sont implicitement quantifiées, existentiellement pour les variables logiques et universellement pour les variables d'indice.

Interprétation des formules Le langage d'assertion introduit des variables logiques, qui ne sont pas présentes dans le états de la sémantique opérationnelle. Nous devons donc étendre la notion d'état :

$$\Sigma = (\sigma, \rho)$$

est un couple composé d'un état σ (au sens de la sémantique opérationnelle), et d'une application ρ de l'ensemble des variables logiques dans celui des valeurs *définies*. L'interprétation des formules se fait de la façon suivante

- si F est une formule close, $\sigma \models F$ s'il existe un ρ tel que $(\sigma, \rho) \models F$;
- pour le prédicat $\Delta : \Sigma = (\sigma, \rho) \models \Delta(X, \vec{\imath}, \alpha)$ si $\sigma(X[\vec{\imath}]) = \rho(\alpha)$; on remarquera que, puisque ρ est une application dans l'ensemble des valeurs définies, ceci implique que X est défini en $\vec{\imath}$;

- pour une formule ne contenant pas le prédicat $\Delta : \Sigma \models F$ ssi $\rho \models F$ au sens usuel de la logique du premier ordre.

Nous avons donc « encapsulé » les deux spécificités de ALPHA (variables polyédriques et présence de la valeur indéfinie) dans le prédicat Δ.

Notations et abus Le prédicat Δ tel qu'il est défini peut s'avérer « lourd » à l'utilisation. Nous introduisons donc les notations suivantes.

- lorsque nous voulons exprimer le fait qu'une composante est définie et vaut une valeur constante a, nous écrirons
$$\Delta(X, \vec{\imath}, a)$$
au lieu de
$$\exists \alpha : \Delta(X, \vec{\imath}, \alpha) \wedge \alpha = a$$

- de même, pour exprimer uniquement le fait qu'une composante est définie, nous écrirons
$$\Delta(X, \vec{\imath}) \text{ ou } \Delta(X[\vec{\imath}])$$
au lieu de
$$\exists \alpha : \Delta(X, \vec{\imath}, \alpha)$$

- nous aurons souvent besoin d'exprimer « si $X[\vec{\imath}]$ est définie, alors sa valeur est celle de l'expression $f(Y_1[\vec{\imath_1}], \ldots, Y_n[\vec{\imath_n}])$ ». Ceci implique en particulier que tous les $Y_i[\vec{\imath_i}]$ sont définies. Nous écrirons donc
$$X[\vec{\imath}] \triangleq f(Y_1[\vec{\imath_1}], \ldots, Y_n[\vec{\imath_n}])$$
pour abréger la formule suivante
$$\exists \alpha : \Delta(X, \vec{\imath}, \alpha) \Rightarrow \exists \beta_1, \ldots, \beta_n : \Delta(Y_1, \vec{\imath_1}, \beta_1) \wedge \ldots \wedge \Delta(Y_n, \vec{\imath_n}, \beta_n) \wedge \alpha = f(\beta_1, \ldots, \beta_n)$$
Remarquons que $\triangleq$ n'est pas un prédicat mais juste une notation.

4.2 Spécifications et validité

Nous voulons définir pour ALPHA une sémantique par assertions inspirée de la logique de Hoare. Nous avons donc besoin d'une notion de formule de spécification, de type $\{P\}\ S\ \{Q\}$. Rappelons qu'une telle formule signifie que, si l'assertion P est vraie dans un état initial, l'exécution de S donne un état satisfaisant l'assertion Q. Nous disposons du langage d'assertion défini dans la section précédente pour exprimer les préconditions P et les postconditions Q. Nous pouvons donc donner la définition suivante

Définition 20 (Formule de spécification) *Une formule de spécification est un triplet, noté*
$$\{P\}\ S\ \{Q\}$$
où S est un programme ALPHA*, et P et Q deux formules du langage d'assertion telles que P ne contient que des variables d'entrée.*

Nous nous limitons cette fois encore à la correction partielle, c'est-à-dire : « si l'exécution donne un état final, alors cet état final satisfait la postcondition ». Alors que dans le cas d'un langage impératif le problème de la terminaison est traité par l'introduction de variants dans la preuve des boucles, dans le cas du langage ALPHA ce problème est davantage un problème d'ordonnancement. Savoir si un état final sera atteint est indécidable ; une condition suffisante est l'existence d'un ordonnancement, qui calcule l'instant d'évaluation de chaque composante. Ces problèmes d'ordonnancement constituent un vaste sujet d'étude qui ne rentre pas dans le cadre de cette thèse.

La sémantique opérationnelle du chapitre précédent nous fournit la notion d'exécution, ainsi que les notions d'état initial et d'état final. Rappelons qu'un état initial est un état dont le support (ensemble des composantes définies) est exactement l'ensemble des composantes d'entrée, et qu'un état final est défini par rapport à un sous-ensemble fini des composantes de sortie. La notion de validité va elle aussi faire appel à cette restriction à un sous-ensemble fini. La définition de celle-ci est la traduction de la définition classique de la validité des triplets de Hoare dans notre formalisme.

Définition 21 (Validité sur un sous-ensemble fini) *Soit $\{P\}\ S\ \{Q\}$ une formule de spécification, et V un sous-ensemble fini de Out. On dit que cette spécification est* valide *par rapport à V, ce qui est noté*

$$\models_V \{P\}\ S\ \{Q\}$$

si pour tout état $\sigma \in Init$ tel que $\sigma \models P$ et pour toute suite de transitions $\sigma \longrightarrow^* \sigma'$ telle que $V \subseteq supp(\sigma')$, nous avons

$$\sigma' \models Q.$$

L'ensemble des composantes de sortie étant potentiellement infini, nous ne pouvons généraliser directement cette définition pour *Out* . Nous pouvons néanmoins définir une notion de validité globale, en « approchant » *Out* par l'ensemble de ses sous-ensembles finis.

Définition 22 (Validité globale) *Soit $\{P\}\ S\ \{Q\}$ une formule de spécification. On dit que cette spécification est* valide, *ce qui est noté*

$$\models \{P\}\ S\ \{Q\},$$

si pour tout sous-ensemble fini V de Out nous avons

$$\models_V \{P\}\ S\ \{Q\}.$$

Nous avons donné une définition de la validité qui s'appuie sur la sémantique opérationnelle de type « flot de données ». Qu'en est-il de la sémantique opérationnelle « à la demande »? Avec cette sémantique, il faut pouvoir « oublier » les transitions concernant des composantes non nécessaires au calcul des sorties. Il faut donc que la postcondition ne dépende pas de ces transitions. Une condition suffisante consiste à imposer la restriction suivante : si deux états sont équivalents sur le sous-ensemble V considéré, alors ils valident en même temps la postcondition. Plus généralement, nous donnons la définition suivante.

Définition 23 (Stabilité) *Soit Q un prédicat, et V un ensemble de composantes. On dit que Q est* stable *sur V si*

$$\forall \sigma, \sigma' : (\sigma \underset{V}{\sim} \sigma' \Rightarrow (\sigma \models Q \ ssi\ \sigma' \models Q)).$$

La spécification $\{P\}\ S\ \{Q\}$ sera dite stable si la postcondition Q est stable.

Il est alors possible de relier les deux notions de validité respectives aux deux sémantiques opérationnelles par la proposition suivante.

Proposition 16 (Validité « à la demande ») *Si Q est un prédicat stable sur un sous-ensemble fini V de Out et*

$$\left.\begin{array}{l}\sigma \in Init\\ \sigma \models P\\ \sigma \underset{V}{\Longrightarrow}^{*} \sigma'\\ supp(\sigma') \cap Out = V\end{array}\right\} \quad \Rightarrow \quad (\sigma' \models Q),$$

alors

$$\models_V \{P\}\ S\ \{Q\}.$$

Preuve. *La preuve est une conséquence directe de la proposition 11 du chapitre 3 et de la définition d'un prédicat stable. Intuitivement : la proposition de réarrangement permet de « repousser » à la fin du calcul les transitions qui ne sont pas nécessaires au calcul de V. Si Q est stable, sa validité dans un état final ne dépend donc pas de ces transitions.* □

4.3 Invariants et prouvabilité

Nous avons défini une notion de validité sur la base de la sémantique opérationnelle. Il nous faut maintenant donner une notion de *prouvabilité* des spécifications. Cette fois, contrairement au cas des langages impératifs, la preuve ne peut se construire grâce à des règles suivant la syntaxe du langage. En effet, le contrôle n'est pas localisable en un point précis du texte du programme. La nature itérative de la règle de transition opérationnelle suggère plutôt d'utiliser des *invariants*. Intuitivement, un invariant est un prédicat qui, s'il est vrai dans un état initial, reste vrai le long de toute suite de transitions partant de cet état initial. La notion de validité d'une spécification a été initialement définie avec la sémantique flot de données, car nous voulions lui donner la définition la plus large possible. La proposition 16 montre que de toute façon une définition équivalente (modulo une hypothèse sur les postconditions) est possible pour la seconde sémantique opérationnelle. En ce qui concerne la notion de prouvabilité, nous ne ferons intervenir que la sémantique à la demande. Nous voudrons en effet au chapitre suivant montrer que notre notion de prouvabilité est « complète », c'est-à-dire que « toute » spécification valide est prouvable. Les détails techniques de la preuve nécessiteront de reconstruire des suites de transitions à partir d'ensembles finis de composantes. C'est pourquoi nous nous restreignons dès la définition de la prouvabilité à des sous-ensembles de variables, donc à la sémantique à la demande.

Définition 24 (Invariant) *Une formule I est un* invariant *pour le programme S, la précondition P et le sous-ensemble $V \subseteq Out$ ssi*

$$\textit{pour tout } \sigma \textit{ tel que } \exists\ \sigma_0 \in Init \textit{ avec } \sigma_0 \models P \textit{ et } \sigma_0 \underset{V}{\Longrightarrow}^{*} \sigma,\ \textit{nous avons } \sigma \models I,$$

où $\underset{V}{\Longrightarrow}$ dénote une transition à la demande relative au programme S et au sous-ensemble V.

Nous pouvons alors donner la définition de la prouvabilité.

Définition 25 (Prouvabilité) *La spécification $\{P\}\ S\ \{Q\}$ est dite* prouvable *par rapport au sous-ensemble fini V de Out, ce qui est noté*

$$\vdash_V \{P\}\ S\ \{Q\},$$

s'il existe un invariant I par rapport à S, P et V tel que

$$\textit{pour tout } \sigma \textit{ tel que } supp(\sigma) \supseteq V, \qquad (\sigma \models I) \Rightarrow (\sigma \models Q).$$

Plus généralement, une spécification $\{P\}\ S\ \{Q\}$ est dite prouvable, *ce qui est noté*

$$\vdash \{P\}\ S\ \{Q\},$$

si elle est prouvable pour tout sous-ensemble fini V de Out.

Par exemple, *true* est un invariant trivial pour toute spécification, mais il ne peut être utilisé pour une preuve dans le cas général, car il ne sera pas possible de prouver que *true* $\Rightarrow Q$ pour tout Q. Revenons à l'exemple du filtre de convolution du chapitre 1. Rappelons tout d'abord le programme.

```
system convolution (a : {j | 1<=j<=4} of integer;
                    x : {i | i>=1} of integer)
returns (y : {i | i>=4} of integer);
var
    Y : {i,j | 0<=j<=4 ; i>=4} of integer;
let
    Y[i,j] = case
             {| j=0 } : 0[];
             {| 1<=j<=4} : Y[i,j-1] + a[j] * x[i-j+1];
             esac;
    y[i] = Y[i,4];
tel;
```

Dans cet exemple, nous avons

$$\begin{aligned} In &= \{a[j] \mid 1 \leq j \leq 4\} \cup \{x[i] \mid 1 \leq i\} \cup \{Y[i,0] \mid 4 \leq i\} \\ Out &= \{y[i] \mid 4 \leq i\} \end{aligned}$$

Dénotons par $|.|$ la fonction valeur absolue. Une spécification pourrait être

$$\begin{aligned} P &\equiv \forall j : (1 \leq j \leq 4) \Rightarrow \exists \alpha : \Delta(a, j, \alpha) \wedge |\alpha| \leq M \\ &\quad \wedge \forall i : (1 \leq i) \Rightarrow \exists \beta : \Delta(x, i, \beta) \wedge |\beta| \leq K \\ Q &\equiv \forall i : (4 \leq i) \Rightarrow \exists \beta : \Delta(Y, i, \beta) \wedge |\beta| \leq 4KM \end{aligned}$$

Fixons une valeur i_0 pour l'indice i. Avec $V = \{y[i_0]\}$, nous avons trivialement

$$\models_V \{P\}\ S\ \{Q\}.$$

Il existe de nombreux invariants. Par exemple, l'un d'eux est

$$\exists \beta : \Delta(y, i_0, \beta) \Rightarrow |\beta| \leq 4KM,$$

mais il n'est pas très intéressant, puisque la seule composante qu'il contient n'est définie que dans un état final. Il ne sera pas utilisable dans une preuve faisant intervenir d'autres composantes. Un autre, plus intéressant, est [1]

$$\begin{array}{ll} & \forall i \geq 1 : \Delta(x[i]) \\ \wedge & \forall j \in \{1 \ldots 4\} : \Delta(a[j]) \\ \wedge & \forall j \in \{1 \ldots 4\} : (Y[i_0, j] \stackrel{\triangle}{=} Y[i_0, j-1] + a[j] * x[i_0 - j + 1]) \\ \wedge & y[i_0] \stackrel{\triangle}{=} Y[i_0, 4] \end{array}$$

Nous verrons dans le chapitre suivant que cet invariant est en quelque sorte l'invariant le plus fort. Nous l'appelerons alors invariant canonique. Auparavant, nous allons établir la correction de notre notion de prouvabilité.

4.4 Correction

Les notions de validité et de prouvabilité des spécifications ont été établies toutes deux sur la base de la sémantique opérationnelle. La correction de la notion de prouvabilité n'est donc pas difficile à établir. Le seul point délicat est le fait que la validité est définie avec la sémantique flot de données alors que la notion d'invariant repose sur la sémantique à la demande. Il faut donc faire intervenir la stabilité. Nous avons immédiatement le théorème suivant.

Théorème 11 (Correction) *Soit V un sous-ensemble fini de Out, et $\{P\}\ S\ \{Q\}$ une spécification stable sur V. Si*

$$\vdash_V \{P\}\ S\ \{Q\}$$

alors

$$\models_V \{P\}\ S\ \{Q\}.$$

Preuve. *D'après la définition de la prouvabilité, il existe une formule I telle que*

1. *pour tout σ tel que $\exists \sigma_0 \in Init : \sigma_0 \models p \quad \wedge \quad \sigma_0 \stackrel{*}{\underset{V}{\Longrightarrow}} \sigma$, on a $\sigma \models I$;*
2. *pour tout σ tel que $V \subseteq supp(\sigma)$, $(\sigma \models I) \Rightarrow (\sigma \models Q)$.*

Prenons une suite de transitions $\sigma \stackrel{*}{\underset{V}{\Longrightarrow}} \sigma'$ telle que $\sigma \in Init$, $\sigma \models P$ et $supp(\sigma') \cap Out = V$. Nous avons alors $V \subseteq supp(\sigma')$, et $\sigma' \models I$. De plus, par (2) nous avons $\sigma' \models I \Rightarrow Q$. Nous avons donc prouvé que

$$\sigma' \models Q.$$

Comme Q est stable sur V, la proposition 16 donne

$$\models_V \{P\}\ S\ \{Q\}.$$

□

1. La définition de l'abréviation « $\stackrel{\triangle}{=}$ » est donnée page 105.

chapitre 5

Une méthodologie de preuve

Nous avons dans le chapitre précédent donné une définition générale de la notion de prouvabilité. Celle-ci repose sur l'existence d'un invariant pour la spécification que l'on veut prouver. Nous n'avons en revanche pas encore donné d'indications sur les méthodes éventuelles pour trouver un invariant potentiel, ni pour prouver qu'une formule est bien un invariant. Ceci constitue le but de ce chapitre.

Une première méthode consisterait à montrer « à la main » qu'une formule est un invariant, en vérifiant qu'elle reste vraie pour toute transition du programme. Plus précisément, il faudrait montrer qu'elle reste vraie pour toute transition à partir d'un état *accessible* du programme. Cette méthode supprime tout l'intérêt du langage logique dont nous disposons, puisqu'elle nous oblige à raisonner une fois de plus « à la main » sur les transitions.

Remarquons par ailleurs le fait suivant : soit I un invariant pour une spécification (et un sous-ensemble de composantes) donnée. Soit I' une autre formule du langage d'assertion, telle que l'on ait

$$\models I \Rightarrow I'$$

Il est alors immédiat de constater que I' est également un invariant.

Nous allons en fait montrer qu'il existe des classes particulières d'invariants, qui seront à la base de notre méthodologie. La première classe est celle de ce que nous appelons les *invariants canoniques*. Ceux-ci consistent en une traduction du comportement opérationnel du programme dans le langage d'assertion. La seconde classe est celle des invariants *univoques*, qui permettent de décrire les sorties d'un programme en fonction des entrées : il s'agit donc de la traduction de la sémantique dénotationnelle dans le langage d'assertion.

5.1 Invariants canoniques

Un invariant canonique est simplement une traduction dans le langage logique du texte d'un programme. Il contient donc toute la sémantique de ce programme, c'est-à-dire à la fois les calculs effectués par celui-ci et l'ordre d'évaluation de ces calculs. De même que la sémantique opérationnelle, il modélise donc tous les ordonnancements possibles d'un programme. L'invariant canonique exprime simplement le fait que

- toute composante d'entrée $A[\vec{\imath}]$ est définie : $\forall \vec{\imath} \in \mathcal{D}_A : \Delta(A[\vec{\imath}])$;
- si une composante $X[\vec{\imath}]$ qui n'est pas une composante d'entrée est définie, alors les composantes dont elle dépend sont définies, et la valeur de la composante est égale à la valeur de son expression de définition : $\forall \vec{\imath} \in \mathcal{D}_X^p : X[\vec{\imath}] \triangleq \varphi(Y[f(\vec{\imath})], Z[g(\vec{\imath})])$.

Au lieu de raisonner sur le texte du programme et de procéder par réécritures, l'invariant canonique va nous permettre de passer dans le « monde » de la logique et de raisonner par inférences et implications. Plus formellement, nous en donnons la définition suivante.

Définition 26 (Invariant canonique) *Soit S un programme* ALPHA. *L'*invariant canonique *de S, noté $I_C(S)$, est la conjonction des formules*

$$\forall \vec{\imath} \in \mathcal{D}_A : \Delta(A[\vec{\imath}])$$

pour toute variable d'entrée A, et des formules

$$\forall \vec{\imath} \in \mathcal{D}_X^p : X[\vec{\imath}] \triangleq \varphi(Y[f(\vec{\imath})], Z[g(\vec{\imath})])$$

pour toute variable X qui n'est pas une variable d'entrée.

L'invariant canonique pour la précondition P *de S est la conjonction*

$$I_C(S) \wedge P$$

Par exemple, si nous reprenons le programme exprimant une propagation de constante donné au chapitre 1, son invariant canonique sera

$$\begin{array}{ll} & \Delta(a) \\ \wedge & X[0] \triangleq a \\ \wedge & \forall t > 0 : X[t] \triangleq X[t-1] \end{array}$$

Nous n'avons pas encore montré que l'invariant canonique était effectivement un invariant. Ceci sera fait dans la section 5.3 où nous avons rassemblé les résultats théoriques. L'invariant canonique ne présente pas un grand intérêt en soi, puisqu'il n'est qu'une réécriture du programme. Son but est de servir de point de départ pour établir la preuve d'autres invariants. D'autre part, il va nous servir pour établir un résultat simple de « complétude » de notre notion de prouvabilité. Mais auparavant, il nous faut définir une autre classe d'invariants.

5.2 Invariants univoques

L'invariant canonique d'un programme est naturellement très simple à produire, mais il peut être trop complexe par rapport à la propriété que nous voulons prouver. Par exemple, nous pouvons être intéressé uniquement par la valeur finale des composantes de sortie, sans nous préoccuper de l'ordre dans lequel celles-ci sont évaluées. Dans ce cas, l'invariant canonique contiendra trop d'« information ». Nous allons donc définir une notion plus « faible » que celle d'invariant canonique : celle d'invariant *univoque*. Plus généralement, une assertion sera dite univoque si elle contient suffisamment d'information pour décrire les valeurs des composantes de sortie d'un programme. Bien entendu, cette notion est définie par rapport à un sous-ensemble fini des composantes de sortie. Nous donnons la définition suivante.

Définition 27 (Assertion univoque) *Soit P une assertion, et V un ensemble des composantes de sortie d'un programme. On dit que P est* univoque *par rapport à V si pour toute paire d'états σ et*

σ', *on a*

$$\left.\begin{array}{c} \sigma \models P \\ \sigma' \models P \\ V \subseteq supp(\sigma) \\ V \subseteq supp(\sigma') \\ \sigma \underset{In}{\sim} \sigma' \end{array}\right\} \quad \Rightarrow \quad \sigma \underset{V}{\sim} \sigma'$$

Si l'on considère un programme, un invariant univoque de ce programme par rapport à un certain sous-ensemble de ses composantes de sortie est donc une assertion qui permet de décrire les valeurs des composantes de sortie après exécution dudit programme. Toujours avec l'exemple de la propagation de constante, un invariant univoque pour l'ensemble total des composantes de sortie et pour la précondition *true* serait

$$\begin{array}{ll} & \Delta(a) \\ \wedge & \forall t \geq 0 : (X[t] \triangleq a) \end{array}$$

Naturellement, un invariant canonique est univoque. Nous en donnons la preuve dans la section suivante. Avant de passer à l'utilisation pour des preuves concrètes de ces notions d'invariants canoniques et univoques, nous allons montrer que notre notion de prouvabilité est complète pour une certaine classe de programmes.

5.3 Complétude restreinte

Nous avons montré dans le chapitre précédent que notre notion de prouvabilité était correcte. Nous allons maintenant montrer la réciproque, c'est-à-dire que celle-ci est complète: pour toute spécification valide, il existe un invariant qui permet de prouver cette spécification. Nous allons naturellement utiliser l'invariant canonique, qui est à la fois le plus simple à exhiber et celui qui contient le plus d'information que l'on puisse tirer d'un programme. Malheureusement, nous ne pourrons obtenir ce résultat pour tous les programmes. En effet, le seul point délicat consiste à montrer que l'invariant canonique implique la postcondition voulue. Pour cela, il va nous falloir construire *explicitement* une suite finie de transitions (voir les détails de la preuve ci-dessous). Il faut donc nous restreindre à un sous-ensemble de programmes ALPHA tels qu'une construction de cette suite soit toujours possible.

Nous donnons la définition suivante.

Définition 28 (Fini clos à gauche) *Soit G le graphe de dépendances d'un programme* ALPHA. *On dit que G est* fini clos à gauche *si pour tout sous-ensemble fini V des composantes de G il existe un sous-ensemble fini W des composantes de G qui contient V et qui est clos à gauche pour la relation de dépendance:*

$$\forall x \in W : \forall y \in G : ((y \preceq x) \Rightarrow (y \in W))$$

Par extension, un programme est fini clos (à gauche) si son graphe de dépendances est fini clos à gauche.

Il est aisé de remarquer que, comme nous avons de plus supposé que le graphe de dépendances était sans cycle (tous les programmes considérés sont bien formés), cette définition correspond exactement à celle d'un programme ordonnançable. La condition imposée revient en effet à dire que toutes les composantes (en particulier les composantes de sortie) doivent être calculables en

un temps fini. Nous conserverons néanmoins cette terminologie de « fini clos », qui aide à mieux comprendre intuitivement le sens des conditions que nous imposerons sur certains programmes. Nous ne pourrons prouver la complétude que pour les programmes vérifiant cette condition de clôture. Cependant, cette restriction n'est pas trop forte, puisque la classe des programmes finis clos contient tous les programmes « utiles ».

Nous aurons besoin pour notre preuve d'utiliser le fait qu'un invariant canonique est univoque. Nous montrons donc au préalable la proposition suivante.

Proposition 17 *Soit S un programme* ALPHA *fini clos. Pour tout sous-ensemble fini V de Out, l'invariant canonique de S est univoque par rapport à V.*

Preuve. *Soit I_C l'invariant canonique de S (pour la précondition true), et supposons qu'il existe un sous-ensemble fini V tel que I_C ne soit pas univoque par rapport à V. Il existe donc deux états σ et σ' tels que*

$$\begin{array}{l} \sigma \models I_C \\ \sigma' \models I_C \\ V \subseteq supp(\sigma) \\ V \subseteq supp(\sigma') \\ \sigma \underset{In}{\sim} \sigma' \end{array} \qquad et \qquad \sigma \not\underset{V}{\sim} \sigma'$$

Remarquons tout d'abord que si $\sigma \models I_C$ et $V \subseteq supp(\sigma)$, alors $\forall x \in Nécessaire(V)$, nous avons $\sigma(x) \neq \perp$ et $\sigma(x) = \sigma(e(\vec{y}))$. Il existe donc une composante x dans V telle que $\sigma(x) \neq \sigma'(x)$, $\sigma(x) \neq \perp$ et $\sigma'(x) \neq \perp$. Comme S est fini clos, tous les chemins menant à x dans le graphe de dépendances sont de longueur finie. Pour toute composante $z \notin In$ telle que $\sigma(z) \neq \sigma'(z)$, il existe une composante $y \preceq z$ telle que $\sigma(y) \neq \sigma'(y)$. Nous en déduisons qu'il existe une composante y dans In, point de départ d'un chemin menant à x, telle que $\sigma(y) \neq \sigma'(y)$, ce qui est impossible. ❑

Nous pouvons maintenant montrer le lien entre invariant canonique et validité d'une spécification.

Théorème 12 (Complétude restreinte) *Soit S un programme* ALPHA *fini clos, $\{P\}\ S\ \{Q\}$ une spécification pour ce programme, et V un sous-ensemble fini de Out. Si*

$$\models_V \{P\}\ S\ \{Q\},$$

alors

$$\vdash_V \{P\}\ S\ \{Q\}.$$

Preuve. *Supposons que $\models_V \{P\}\ S\ \{Q\}$, et soit I l'invariant canonique de S pour P : $I = I_C(S) \wedge P$. Nous devons prouver*

1. *pour tout σ tel que $\exists \sigma_0 \in Init, \sigma_0 \models P$ et $\sigma_0 \underset{V}{\Longrightarrow}^* \sigma$, $\quad \sigma \models I$;*
2. *pour tout σ tel que $V = supp(\sigma) \cap Out$: si $\sigma \models I$, alors $\sigma \models Q$.*

Le premier point est prouvé par récurrence sur la longueur n de la suite de transitions $\sigma_0 \underset{V}{\Longrightarrow}^n \sigma$. Si $n = 0$, nous avons $\sigma = \sigma_0$. Comme $\sigma_0 \models P$ et σ est indéfini sur les composantes qui ne sont pas des composantes d'entrée, nous avons $\sigma \models I$. Supposons maintenant que le résultat a été établi pour $n-1$ et considérons une suite $\sigma_0 \underset{V}{\Longrightarrow}^n \sigma$. Il existe une composante x et un

état σ' tels que $\sigma_0 \underset{V}{\Longrightarrow}^{n-1} \sigma' \underset{V}{\overset{x}{\Longrightarrow}} \sigma$. Par hypothèse de récurrence, nous savons que $\sigma' \models I$. La dernière transition est valide, donc $\sigma'(x) = \perp$ et $\sigma(x) = \sigma'(e(\vec{y}))$. La composante x correspond à une certaine variable X et un certain indice $\vec{\imath}$. Dans σ nous avons $\Delta(X[\vec{\imath}]) = true$. Comme $\sigma(x) = \sigma'(e(\vec{y}))$, nous avons $\sigma \models X[\vec{\imath}] = e(\vec{Y}[\vec{f}(\vec{\imath})])$. Dans σ' nous avons $\Delta(\vec{Y}[\vec{f}(\vec{\imath})]) = true$, donc ceci reste vrai dans σ. De plus, $\Delta(X[\vec{\imath}])$ est devenu vrai dans σ : nous concluons que

$$\sigma \models \Delta(X[\vec{\imath}]) \Rightarrow \Delta(\vec{Y}[\vec{f}(\vec{\imath})]).$$

Finalement, comme P ne contient que des composantes d'entrée, il n'est pas modifié par la transition.

Prouvons maintenant le second point. Considérons un état σ tel que $V = supp(\sigma) \cap Out$ et $\sigma \models I$. Soit σ_1 la restriction de σ à In, qui est dénotée par $\sigma_1 = \sigma\,|_{In}$: nous avons $supp(\sigma_1) = In$ et $\sigma_1 \underset{In}{\sim} \sigma$. Comme $\sigma \models I$, nous savons que $\sigma_1 \models P$. Soit maintenant σ_1' tel que $V \subseteq supp(\sigma_1')$ et

$$\sigma_1 \underset{V}{\Longrightarrow}^{*} \sigma_1'.$$

Un tel σ_1' existe car S est fini clos. De plus, σ_1' satisfait l'invariant canonique. Comme cet invariant est univoque d'après la proposition 17, nous concluons que $\sigma_1'|_V = \sigma'|_V$. Finalement, comme la spécification $\{P\}\ S\ \{Q\}$ est valide, $\sigma_1' \models Q$, donc

$$\sigma' \models Q.$$

□

5.4 Preuve de propriétés partielles

Même si les invariants canoniques contiennent trop d'information, ils vont nous être utiles pour prouver des propriétés partielles, c'est-à-dire des propriétés ne requérant pas une connaissance complète des valeurs des composantes. Nous allons en effet utiliser le fait, mentionné dans l'introduction de ce chapitre, que si I est un invariant et si $\models I \Rightarrow I'$, alors I' est également un invariant. Dans une telle implication, c'est justement l'invariant canonique qui va nous servir de point de départ. Il suffit donc de prouver que la formule que nous voulons établir est impliquée par l'invariant canonique du programme considéré.

Revenons à l'exemple de la convolution du chapitre 1. Dans cet exemple, nous avons

$$\begin{aligned} In &= \{a[j] \mid 1 \leq j \leq 4\} \cup \{x[i] \mid 1 \leq i\} \cup \{Y[i,0] \mid 4 \leq i\} \\ Out &= \{y[i] \mid 4 \leq i\} \end{aligned}$$

Fixons une valeur $i_0 \geq 4$ pour i, et dénotons par $|.|$ la fonction valeur absolue. Une spécification pourrait être

$$\begin{aligned} P &\equiv \forall j \in \{1 \ldots 4\} \Rightarrow \exists \alpha : \Delta(a, j, \alpha) \wedge |\alpha| \leq M \wedge \exists \beta : \Delta(x, i_0 + j - 1, \beta) \wedge |\beta| \leq K \\ Q &\equiv \exists \beta : \Delta(Y, i_0, \beta) \wedge |\beta| \leq 4KM \end{aligned}$$

En prenant $V = \{y[i_0]\}$, il est facile de constater que $\{P\}\ S\ \{Q\}$ est valide par rapport à V.

L'invariant canonique I_C de ce programme pour cette spécification est

$$\begin{aligned} & \forall i \geq 4 : \Delta(x[i]) \ \wedge \ \forall j \in \{1 \ldots 4\} : \Delta(a[j]) \\ \wedge \quad & \forall i \geq 4 : \forall j \in \{1 \ldots 4\} : (Y[i,j] \triangleq Y[i,j-1] + a[j] * x[i-j+1]) \\ \wedge \quad & \forall i \geq 4 : y[i] \triangleq Y[i,4] \end{aligned}$$

Soit maintenant I la formule suivante

$$\begin{aligned} & \forall j \in \{1 \ldots 4\} : \exists \gamma : \Delta(Y, (i_0, j), \gamma) \Rightarrow (|\gamma| \leq jKM) \\ \wedge \quad & y[i_0] \triangleq Y[i_0, 4] \end{aligned}$$

Il est facile de montrer, par induction sur j, que $I_C \Rightarrow I$. Donc I est un invariant acceptable. De plus, il implique la postcondition voulue.

Sur cet exemple simple, la preuve aurait naturellement pu être faite « à la main ». Cependant, le raisonnement à partir d'invariants permet d'une part de respecter un cadre logique qui garantit la validité du résultat et fournit une méthodologie applicable à des exemples non triviaux, et d'autre part d'envisager une automatisation (au moins partielle) de telles preuves[1].

5.5 Preuve d'équivalences

Nous avons vu que, de façon générale, l'invariant canonique peut servir de point de départ pour la preuve de toute propriété, même partielle. Nous allons voir maintenant que les invariants univoques sont particulièrement adaptés à la preuve d'équivalence de programmes. Nous dirons que deux programmes ont équivalents s'ils fournissent les mêmes valeurs de sortie à partir des mêmes valeurs d'entrée. Plus précisément, nous donnons la définition suivante.

Définition 29 (Programmes équivalents) *Soit P une assertion sur des composantes d'entrée, et V un sous-ensemble fini des composantes de sortie. On dit que deux programmes sont* équivalents à partir de P par rapport à V *s'ils ont les mêmes ensembles de composantes d'entrée et de sortie, et si, partant du même état initial satisfaisant P, ils arrivent dans des états finaux (par rapport à V) qui sont équivalents modulo V.*

Si P est l'assertion true, on dit simplement que les programmes sont équivalents par rapport à V.

Intuitivement, la notion d'invariant univoque signifie qu'un tel invariant contient suffisamment d'information pour décrire les résultats du programme. Deux programmes respectant le même invariant univoque seront donc équivalents. Ceci est exprimé par la proposition suivante.

Proposition 18 *Soit S et S' deux programmes ayant les mêmes ensembles de composantes d'entrée et de sortie In et Out, soit P une assertion sur les composantes d'entrée, et soit V un sous-ensemble fini de Out. Si S et S' admettent un invariant univoque commun par rapport à P et V, alors S et S' sont équivalents à partir de P par rapport à V.*

La preuve de cette proposition est immédiate, d'après la définition d'un invariant univoque. Comme dans le cas des propriétés partielles, vérifier que deux programmes sont équivalents se fera

1. Ce point sera discuté plus précisément à la fin de ce chapitre.

par raisonnement sur les invariants. Il suffit de leur trouver un invariant univoque commun. Or nous savons que les invariants canoniques sont univoques. On pourra donc

- soit prouver que l'invariant canonique d'un programme implique l'invariant canonique de l'autre programme,

- soit prouver que les deux invariants canoniques impliquent un même invariant univoque, plus « simple »,

ou utiliser toute combinaison de ces stratégies avec des invariants univoques intermédiaires.

Prenons par exemple le problème du calcul de la fonction de temps exposé au chapitre 1. Rappelons les programmes concernés. Le système qui calcule l'ordonnancement au plus tôt est le suivant

```
system Free : (N: { N | N>=2 } parameter)
returns (free : { i,j,k | 0<=k<=N; 0<i,j<=N } of integer);
var idxk : { x |} of integer;
let
    idxk[x] = case
        {|x<0} : idxk[x+1] - 1[];
        {|x=0} : 0[];
        {|x>0} : idxk[x-1] + 1[];
    esac;
    free[i,j,k] = case
            {| k=0} : 0[];
            {| k>0; i=j=k} : 1[] + free[i,j,k-1];
            {| k>0; j=k; i>k} | {| k>0; j=k; i<k} :
                    1[] + (free[i,j,k-1] max free[k,j,k]);
            {| k>0; i=k; j>k} | {| k>0; i=k; j<k} :
                    1[] + (free[i,k,k] max free[i,j,k-1]);
            {| k>0; i>k>j} | {| k>0; j>k>i} | {| k>0; j,i>k} | {| k>0; i,j<k} :
                    1[] + (free[i,j,k-1] max free[i,k,k] max free[k,j,k]);
                esac;
tel;
```

Nous voulons prouver l'équivalence avec le système suivant, qui donne le même résultat sur les sorties, mais avec un calcul direct (non récursif). Nous avons renommé les variables du système afin

d'avoir les mêmes noms pour les variables de sortie.

```
system Closed : (N:{ N | N>=2 } parameter)
returns (free : { i,j,k | 0<=k<=N; 0<i,j<=N } of integer);
var idxk : { x |} of integer;
let
    idxk[x] = case
        {|x<0} : idxk[x+1] - 1[];
        {|x=0} : 0[];
        {|x>0} : idxk[x-1] + 1[];
    esac;
    free[i,j,k] = case
            {| k=0} : 0[];
            {| k>0; i=j=k} : 3[]*idxk[k] - 2[];
            {| k>0; j=k; i>k} | {| k>0; j=k; i<k}
                | {| k>0; i=k; j>k} | {| k>0; i=k; j<k} : 3[]*idxk[k] - 1[];
            {| k>0; i>k>j} | {| k>0; j>k>i}
                | {| k>0; j,i>k} | {| k>0; i,j<k} : 3[]*idxk[k];
                    esac;
tel;
```

Considérons la formule suivante :

$$
\begin{array}{rcl}
I & \equiv & \forall k : 0 \leq k \leq N \Rightarrow idxk[k] \triangleq k \\
& \wedge & \forall i : \forall j : \forall k : \quad k = 0 \Rightarrow free[i,j,k] \triangleq 0 \\
& & \qquad \wedge \quad (k > 0 \wedge i = j = k) \Rightarrow free[i,j,k] \triangleq 3 * k - 2 \\
& & \qquad \wedge \quad ((k > 0 \wedge j = k \wedge i > k) \vee (k > 0 \wedge j = k \wedge i < k) \\
& & \qquad\quad \vee (k > 0 \wedge j < k \wedge i = k) \vee (k > 0 \wedge j > k \wedge i = k)) \\
& & \qquad\quad \Rightarrow free[i,j,k] \triangleq 3 * k - 1 \\
& & \qquad \wedge \quad ((k > 0 \wedge i > k > j) \vee (k > 0 \wedge j > k > i) \\
& & \qquad\quad \vee (k > 0 \wedge j > k \wedge i > k) \vee (k > 0 \wedge j < k \wedge i < k)) \\
& & \qquad\quad \Rightarrow free[i,j,k] \triangleq 3 * k
\end{array}
$$

Il est facile de montrer que cette formule est un invariant pour le second programme. Il suffit de raisonner par implication à partir de son invariant canonique. Le premier terme se montre par une récurrence sur k, le reste par une simple réécriture de *idxk*. De plus, cet invariant est univoque, puisqu'il décrit totalement les valeurs des composantes de sortie — qui sont de plus indépendantes de toute entrée.

Il nous faut maintenant prouver que I est un invariant pour le premier programme. Nous allons donc montrer que l'invariant canonique du premier programme implique I. Ce point est plus délicat. Il nous faut en effet construire une nouvelle preuve par récurrence. Dans la preuve de $idxk[k] = k$, l'« idée » de procéder par récurrence sur k est immédiate. Dans ce nouveau cas, nous avons le choix entre une preuve « à la main », c'est-à-dire qui examine une à une toutes les transitions possibles, ou une preuve par récurrence un peu plus structurée, qui les examine par « paquets ». Nous avons alors besoin de connaître l'instant de calcul de chaque composante, c'est-à-dire un ordonnancement du programme. La tâche ici n'est pas trop complexe, puisque justement le second programme calcule une fonction de temps potentielle. Il nous suffit d'adopter cette fonction et de nous en servir pour

notre récurrence. Posons donc

$$\varphi(i,j,k) = \begin{cases} 0 & \text{si} \quad k = 0 \\ 3k-2 & \text{si} \quad k > 0 \wedge i = j = k \\ 3k-1 & \text{si} \quad (k > 0 \wedge j = k \wedge i > k) \vee (k > 0 \wedge j = k \wedge i < k) \\ & \quad \vee (k > 0 \wedge j < k \wedge i = k) \vee (k > 0 \wedge j > k \wedge i = k) \\ 3k & \text{si} \quad (k > 0 \wedge i > k > j) \vee (k > 0 \wedge j > k > i) \\ & \quad \vee (k > 0 \wedge j > k \wedge i > k) \vee (k > 0 \wedge j < k \wedge i < k) \end{cases}$$

Nous n'avons pas besoin ici de prouver que φ est une fonction de temps optimale, nous avons juste besoin de vérifier que cette fonction respecte les dépendances du premier programme. Cette vérification est purement mécanique. La preuve de l'implication entre l'invariant canonique et I ne pose maintenant plus de problème : nous pouvons l'établir par récurrence sur $\varphi(i,j,k)$.

Cet exemple d'équivalence de programmes met en évidence plusieurs points essentiels :

- Nous disposons d'un cadre formel qui permet d'établir des propriétés — en l'occurrence des équivalences de programmes — de façon fondée. La méthode de preuve consiste à raisonner par implication entre des invariants. Lorsque ces implications font intervenir des preuves par récurrence, celles-ci doivent tenir compte des dépendances du programme. La méthode de preuve par invariant est donc étroitement liée à des aspects plus « traditionnels » du langage ALPHA, tels que les problèmes d'ordonnancement. Ce point sera discuté à la fin de ce chapitre.

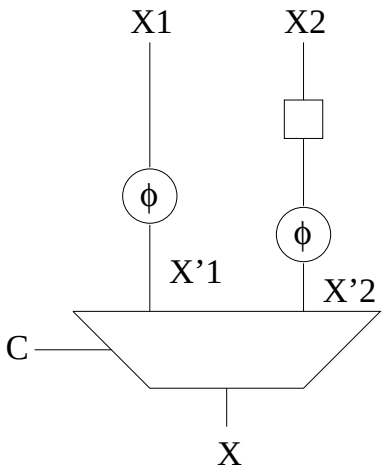
- L'exemple que nous avons pris ici est un peu particulier. En effet, le second programme n'en est pas véritablement un. C'est plutôt un moyen, dans l'article initial ([62]), d'exprimer la propriété que nous avons ici codée par I. Cet exemple nous montre donc à la fois comment traiter le problème d'équivalence de programmes faisant intervenir des calculs par récurrence, ou plus généralement comment prouver qu'une formule est un invariant univoque d'un programme.

5.6 Exemple

Nous allons reprendre l'exemple du circuit avec multiplexeur donné au chapitre 1. Le circuit initial et le programme ALPHA correspondant sont donnés en figure 5.1. L'entrée C du multiplexeur détermine laquelle des deux entrées X_1 ou X_2 sera dirigée sur la sortie. Ces deux entrées sont préalablement traitées par le même opérateur ϕ (une multiplication par 2 dans le texte du programme).

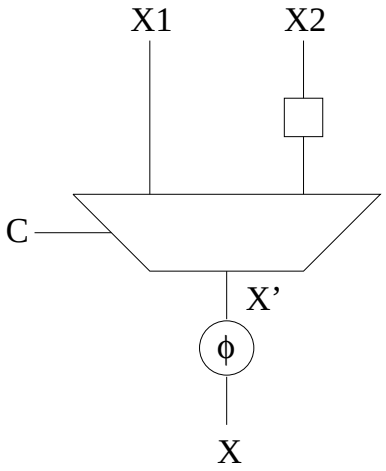
Nous donnons ensuite une autre « version » de ce circuit, où l'opérateur ϕ n'est plus dupliqué mais partagé, en le plaçant après le multiplexeur. Ce nouveau circuit et le programme correspondant sont donnés en figure 5.2. Une telle transformation est appelée *retiming*.

Dans un premier temps, nous allons montrer que ces deux circuits sont équivalents. Nous considérons donc les invariants canoniques des deux programmes par rapport au sous-ensemble de composantes $V = \{X[i] \mid 0 \leq i \leq N+1\}$ et à la précondition $true$, que nous notons respectivement I_1




```
system hardware_share1:
   (N: {N|N>=0} parameter
   X1,X2: {t| t>=0;t<=N+1} of integer;
   C:{t|t>=0} of boolean)
returns (X: {t|t>=0;t<=N+1} of integer);
var
   X1prim: {t| t>=0;t<=N+1} of integer;
   X2prim: {t| t>=1;t<=N+1} of integer;
let
   X[t] = if C[t] then
          case {|0<=t<=N}: X1prim[t];
               {|t=N+1}: 0[];
          esac else
          case {|1<=t<=N+1}: X2prim[t];
               {|t=0}: 0[];
          esac;
   X1prim[t] = X1[t] * 2[];
   X2prim[t] = X2[t-1] * 2[];
tel;
```

FIG. 5.1 – *Un exemple de circuit (multiplexeur)*




```
system hardware_share2:
   (N: {N|N>=0} parameter
   X1,X2: {t| t>=0;t<=N+1} of integer;
   C:{t|t>=0} of boolean)
returns (X: {t|t>=0;t<=N+1} of integer);
var
   Xprim: {t| t>=0;t<=N+1} of integer;
let
   X[t] = Xprim[t]*2[];
   Xprim[t] = if C[t] then
          case {|0<=t<=N}: X1[t];
               {|t=N+1}: 0[];
          esac else
          case {|1<=t<=N+1}: X2[t-1];
               {|t=0}: 0[];
          esac;
tel;
```

FIG. 5.2 – *Le circuit après* retiming

et I_2. Nous avons

$$\begin{aligned} I_1 \quad \equiv \quad & \forall t \in \{0,..,N+1\} : \Delta(X_1[t]) \wedge \Delta(X_2[t]) \\ \wedge \quad & \forall t \geq 0 : \Delta(C[t]) \\ \wedge \quad & \forall t \in \{0,..,N+1\} : X_1'[t] \triangleq 2 * X_1[t] \\ \wedge \quad & \forall t \in \{1,..,N+1\} : X_2'[t] \triangleq 2 * X_2[t-1] \\ \wedge \quad & \forall t \in \{1,..,N\} : X[t] \triangleq (C[t]?X_1'[t] : X_2'[t]) \\ \wedge \quad & X[0] \triangleq (C[0]?X_1'[0] : 0) \\ \wedge \quad & X[N+1] \triangleq (C[N+1]?0 : X_2'[N+1]) \end{aligned}$$

et

$$\begin{aligned} I_2 \quad \equiv \quad & \forall t \in \{0,..,N+1\} : \Delta(X_1[t]) \wedge \Delta(X_2[t]) \\ \wedge \quad & \forall t \geq 0 : \Delta(C[t]) \\ \wedge \quad & \forall t \in \{0,..,N+1\} : X[t] \triangleq 2 * X'[t] \\ \wedge \quad & \forall t \in \{1,..,N\} : X'[t] \triangleq (C[t]?X_1[t] : X_2[t-1]) \\ \wedge \quad & X'[0] \triangleq (C[0]?X_1[0] : 0) \\ \wedge \quad & X'[N+1] \triangleq (C[N+1]?0 : X_2[N]) \end{aligned}$$

Soit maintenant I la formule suivante.

$$\begin{aligned} I \quad \equiv \quad & \forall t \in \{1,..,N\} : \ X[t] \triangleq (C[t]?2 * X_1[t] : 2 * X_2[t-1]) \\ \wedge \quad & X[0] \triangleq (C[0]?2 * X_1[0] : 0) \\ \wedge \quad & X[N+1] \triangleq (C[N+1]?0 : 2 * X_2[N]) \end{aligned}$$

Il est immédiat de montrer, par une analyse de cas sur t, que

$$I_1 \Rightarrow I \text{ et } I_2 \Rightarrow I.$$

Donc I est un invariant pour les deux programmes. De plus, I définit précisément les valeurs des sorties en fonction des valeurs des entrées, donc il est univoque. Nous pouvons en conclure que les deux programmes sont équivalents.

Maintenant, spécifions le contrôle C de la façon suivante.

```
C[t] = case
        {| t = 0 } : true;
        {| t >0 } : not C[t-1];
                esac;
```

Nous pouvons montrer, par récurrence sur t, que ceci revient à prendre la précondition

$$P \equiv \forall t \geq 0 : \Delta(C, 2 * t, true) \wedge \Delta(C, 2 * t + 1, false)$$

Avec cette précondition, l'invariant canonique du premier programme devient

$$\begin{aligned} I_1 \equiv\ & \forall t \geq 0 : \Delta(C, 2*t, true) \wedge \Delta(C, 2*t+1, false) \\ \wedge\ & \forall t \in \{0,..,N+1\} : \Delta(X_1[t]) \wedge \Delta(X_2[t]) \\ \wedge\ & \forall t \in \{0,..,N+1\} : X_1'[t] \triangleq 2 * X_1[t] \\ \wedge\ & \forall t \in \{1,..,N+1\} : X_2'[t] \triangleq 2 * X_2[t-1] \\ \wedge\ & \forall t \in \{1,..,N\} : X[t] \triangleq (C[t]?X_1'[t] : X_2'[t]) \\ \wedge\ & X[0] \triangleq (C[0]?X_1'[0] : 0) \\ \wedge\ & X[N+1] \triangleq (C[N+1]?0 : X_2'[N+1]) \end{aligned}$$

Comme précédemment, nous avons

$$I_1 \Rightarrow I$$

où I est modifié de la même façon :

$$\begin{aligned} I \equiv\ & \forall t \geq 0 : \Delta(C, 2*t, true) \wedge \Delta(C, 2*t+1, false) \\ \wedge\ & X[0] \triangleq (C[0]?2 * X_1[0] : 0) \\ \wedge\ & X[N+1] \triangleq (C[N+1]?0 : 2 * X_2[N]) \\ \wedge\ & \forall t \in \{1,..,N\} : X[t] \triangleq (C[t]?2 * X_1[t] : 2 * X_2[t-1]) \end{aligned}$$

Il est finalement facile de montrer que $I \Rightarrow I'$, avec

$$\begin{aligned} I' \equiv\ & \forall t \in \{0,..,\lfloor N/2 \rfloor\} :\ X[2*t] \triangleq 2 * X_1[2*t] \\ & \wedge X[2*t+1] \triangleq 2 * X_2[2*t] \end{aligned}$$

5.7 Discussion

À partir du cadre logique que nous avons proposé dans le chapitre précédent, nous avons donné ici une première méthodologie de preuve pour les programmes ALPHA. Nous avons ainsi mis en évidence différents types d'invariants. Les invariants canoniques sont une traduction dans notre cadre du programme dans son entier. Les invariants univoques permettent quant à eux de décrire les valeurs des sorties. La méthodologie proposée consiste à procéder par implications entre des invariants, en partant de l'invariant canonique et en l'affaiblissant, si nécessaire en plusieurs étapes. Le cadre logique et la méthodologie proposés garantissent la validité des preuves effectuées.

Nous avons passé en revue différents types de propriétés à prouver. La méthodologie que nous présentons permet de prouver des propriétés partielles, des propriétés d'équivalence de programmes, et des propriétés non exprimables en ALPHA. Un exemple particulièrement intéressant est celui de propriétés faisant intervenir des preuves par induction. Un tel résultat ne peut être obtenu si l'on reste dans le système ALPHA en procédant par réécritures.

Se pose maintenant le problème de l'implantation de cette méthodologie. Le but du langage ALPHA tel qu'il est actuellement développé est de pouvoir passer de la façon la plus automatique possible d'une spécification haut-niveau à une description de circuit. L'implantation du cadre de preuve doit se rapprocher le plus possible du système ALPHA. D'une part pour que les propriétés prouvées soient « utiles » au système, et d'autre part parce que la preuve elle-même doit tirer

parti des fonctionnalités du système ALPHA. Nous avons vu par exemple que certaines propriétés nécessitaient des preuves par récurrence. Il faut dans ce cas être capable de préciser sur quel « ordre » se fait la récurrence. Bien souvent, cette indication nous est donnée par un ordonnancement du programme. Plus généralement, l'approche proposée ici est « tout logique » : le programme est traduit en formules, et l'on raisonne ensuite uniquement au niveau de la logique, sans pouvoir utiliser la structure du langage ALPHA. Une approche plus complète serait, de la même façon que dans les langages impératifs les formules logiques sont insérées dans le texte du programme sous forme d'annotations, d'enrichir la syntaxe avec de la logique. Ceci permettrait de tirer parti à la fois de la structure syntaxique du programme, et de la richesse supplémentaire du langage logique. La similitude entre l'invariant canonique et les équations du programme (il suffit de changer le '=' équationnel en '$\stackrel{\Delta}{=}$' « définitionnel » semble indiquer qu'une telle voie est possible.

chapitre 6

Vers la modularité

Le langage ALPHA tel qu'il a été défini initialement par Mauras [52] ne peut être utilisé pour construire des applications de taille importante. Outre les opérateurs de réduction introduits par Le Verge [48], un premier ajout important au langage a été fait par Saouter ([63]) et de Dinechin ([30]), qui ont introduit des *paramètres* respectivement dans la définition des systèmes d'équations récurrentes et dans le langage ALPHA lui-même. Le langage que nous avons utilisé jusqu'ici tenait compte de cet ajout au langage initial. Nous n'insistons pas ici sur cette paramétrisation : les problèmes induits par leur introduction concernent davantage l'ordonnancement et l'implémentation du système ALPHA que le cadre logique qui nous concerne.

La paramétrisation des systèmes permet une certaine réutilisabilité de ceux-ci, mais ce n'est pas suffisant. Pour permettre la conception de circuits de taille respectable, une structuration du langage a été introduite par de Dinechin [30]. Celle-ci permet de concevoir les spécifications de façon *hiérarchique*, en découpant le problème en modules. Ceci rend les programmes ALPHA beaucoup plus lisibles, puisqu'ils respectent ainsi un découpage « naturel » des fonctions décrites, et permet en outre de limiter la dimension des variables traitées.

Dans ce chapitre, nous allons présenter rapidement cette structuration telle qu'elle a été définie par de Dinechin, puis nous essaierons de donner une première piste pour élargir notre cadre logique en intégrant cette extension.

6.1 Modules et extension de dimension

Le but de la structuration est de pouvoir construire des sous-systèmes qui seront utilisés comme « briques » pour définir des systèmes plus complexes. Un sous-système ou module sera donc un système ALPHA, avec des arguments d'entrée et de sortie qui permettent de l'interfacer avec le système englobant. La structuration doit permettre de « déléguer » le calcul à une sous-unité du programme, comme le ferait un appel de procédure, mais elle doit également permettre de « dupliquer » les appels à ces sous-unités. Considérons par exemple une application devant effectuer des produits de matrices. Il devrait être possible d'écrire une fois pour toutes un système calculant un produit de matrices. Ce système sera paramétré par les dimensions des matrices, et pourra alors être appelé par notre application. Considérons maintenant que notre application décrit par exemple un traitement de signal, et qu'elle doit effectuer une certaine quantité de produits de matrices à chaque unité de temps. Comme il n'est pas possible d'écrire une boucle en ALPHA, ce problème va être traité par l'ajout d'une *extension de dimension*, qui va permettre de répéter, spatialement ou temporellement selon l'interprétation choisie, l'appel au sous-système. Prenons un exemple. Le système décrit en figure 6.1, tiré de la thèse de de Dinechin [30], calcule une addition de deux

nombres binaires de taille W. Il utilise pour cela une cellule `FullAdder` classique, décrite par un sous système prenant en entrée trois booléens et en retournant deux.

```
system Plus : { W | W >1 } ( A, B : { b | 0<=b<W } of boolean )
        returns ( S : { b | 0<=b<W } of boolean );
var Cin, Cout, X : { b | 0<=b<W } of boolean;
let
    use { b | 0<=b<W } FullAdder (A,B,Cin) returns (X,Cout);
    Cin[b] = case
            {| b=0 }: 0[];
            {| b>0 }: Cout[b-1];
                esac;
    S[b] = case
            {| b<W }: X[b];
            {| b=W }: Cout[W-1];
                esac;
tel;
```

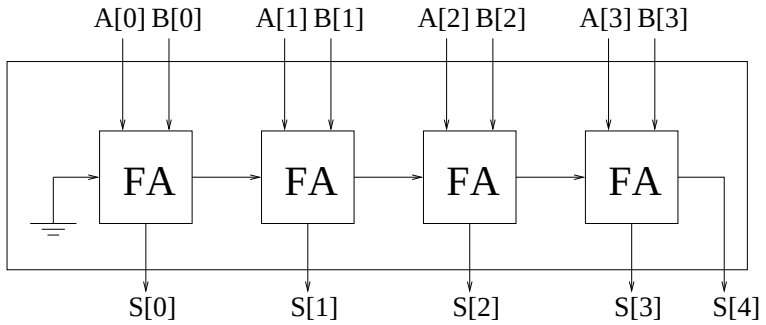


FIG. 6.1 – *Système calculant une addition de nombres binaires de taille W*

Le système décrit en figure 6.2 calcule une multiplication par l'algorithme classique, en utilisant le système d'addition précédent. Les entrées et sorties sont donc des vecteurs de booléens.

Dans ces deux systèmes, la duplication des sous-systèmes se fait par l'indication d'un domaine ALPHA placé juste après le mot-clef `use`. Le système `Plus` fait appel à W cellules `FullAdder`, et le système `Times` à $W-1$ additionneurs. Dans les deux cas, il n'est pas précisé si cette duplication est spatiale (une recopie physique du circuit) ou temporelle (une boucle qui utilise plusieurs fois la ressource). C'est uniquement l'interprétation finale des indices qui décide de ce point.

Nous ne détaillerons pas ici les problèmes de calcul des domaines et de renommage des variables dus aux extensions de dimension. Ceux-ci sont traités dans la thèse de de Dinechin [30]. Signalons seulement que le domaine de l'argument *réel* (celui de l'appel du `use`) est le produit (dans cet ordre) du domaine de l'argument formel (celui de la déclaration du sous-système) par le domaine de l'extension de dimension. Les indices de l'extension sont donc ajoutés à droite par rapport aux indices « internes » au module. Par exemple, dans le système `Times`, les variables d'entrée du système `Plus` sont bidimensionelles, car l'appel à ce système se fait pour $W-1$ valeurs de m (ce qui correspond à la seconde dimension desdites variables).

```
system Times : { W | W>2 } ( A, B : { b | 0<=b<W } of boolean )
        returns ( X : { b | 0<=b<W } of boolean );
var
    P : { b,m | 0<=b<W; 0<=m<W } of boolean;
    Si : { b,m | 0<=b<W; 0<m<W } of boolean;
    So : { b,m | 0<=b<=W; 0<m<W } of boolean;
let
    P[b,m] = A[b] and B[m];
    use { m | 0<m<W } Plus[W] (Si,P) returns (So);
    Si[b,m] = case
            {| m=1 } : P[b,m-1];
            {| m>1 } : So[b+1,m-1];
                esac;
    X[b] = So[b+1,W-1];
tel;
```

FIG. 6.2 – *Système calculant une multiplication de nombres binaires de taille W*

6.2 Substitution

La sémantique initiale donnée pour les modules est une *sémantique par substitution* : un système avec modules est équivalent à ce système dans lequel les modules ont été remplacés par les équations les composant, après renommage et extension de dimension des variables du module. Nous ne donnons pas ici les détails techniques de ces renommages et extensions (il faut par exemple étendre les fonctions affines de dépendance des équations). Nous supposerons donc pour la suite que ceux-ci ont déjà été effectués. Un module pour nous est donc un sous-système dont les variables et les équations ont été étendues par le domaine de l'extension, et dont les variables d'entrée et de sortie ont été renommées pour correspondre aux noms des arguments réels. Plusieurs `use` appelant le même sous-système avec des extensions de dimension et des arguments différents donneront donc des modules différents.

Cette sémantique par substitution pose des problèmes au niveau de la modularité. En effet, l'appel à un module ne peut être vu comme un appel de procédure : il n'est pas nécessaire d'attendre que toutes les entrées soient définies pour commencer les calculs internes au module. Reprenons l'exemple de la multiplication : il n'est pas nécessaire d'attendre que tous les bits du résultat de la première addition soient calculés pour commencer l'addition suivante. Dans certains cas, ce sera même impossible, si le domaine d'un argument d'entrée a une composante infinie par exemple. Nous allons voir dans la section suivante comment gérer cet aspect. Auparavant, nous introduisons les notations suivantes.

Notations et premières hypothèses Dans toute la suite, nous considérerons un système S, faisant appel à *un seul* module S' (cf. figure 6.3). Nous noterons

- $\overline{S}$ le programme S dans lequel on a substitué l'appel au module par le module lui-même, après renommage et extension ; $\overline{S}$ est défini à renommage près des variables locales au module ;
- $In(S), Out(S), Loc(S), Var(S)$ les ensembles respectifs de composantes d'entrée, de sortie,

locales de S, et l'union de ces ensembles ;

- de même pour $In(S'), \ldots$ et $In(\overline{S}), \ldots$, que nous noterons également $In', \ldots$ et $\overline{In}, \ldots$; on a $In(\overline{S}) = In(S), Out(\overline{S}) = Out(S)$ et $Var(\overline{S}) = Var(S) \cup Loc(S')$.

Le graphe de dépendance de $\overline{S}$ contient d'une part les dépendances de S, d'autre part les dépendances directes

$$y \to x$$

pour tout couple (x, y) de $In' \times Out'$ tel qu'il existe un chemin de x à y dans le graphe de dépendances de S'. *Nous supposerons qu'il n'y a pas de dépendances entre des composantes de* Out'.

Nous supposerons de plus que les ensembles de sortie de S et de son module sont disjoints

$$Out \cap Out' = \emptyset$$

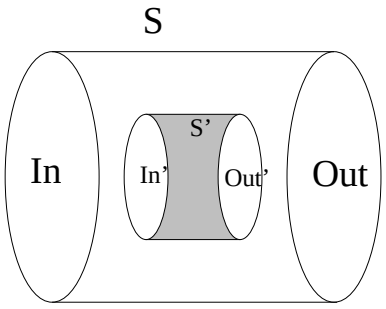


FIG. 6.3 – *Un système avec un module*

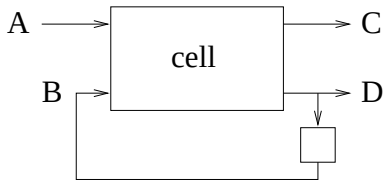
6.3 Sémantique opérationnelle

Nous allons tenter dans cette section de donner une sémantique opérationnelle aux programmes ALPHA modulaires. Cette sémantique doit être elle-même modulaire, c'est-à-dire *compositionnelle* : nous devons pouvoir exprimer la sémantique d'un système en fonction des sémantiques de ses constituants. Il faut donc « voir » de la même façon un système et un sous-système lui appartenant. Par sa nature même, la sémantique par substitution n'est pas modulaire. De plus, le problème des domaines d'entrée éventuellement infinis interdit de généraliser directement la sémantique opérationnelle donnée pour les programmes sans modules. En effet, cette sémantique considérait que dans un état initial, toutes les composantes d'entrée sont valuées. Si ce nombre est infini, comment à partir d'un état initial du système pourrait-on arriver en un nombre fini de transitions à un état initial du module?

Un autre problème se pose, qui est celui des éventuelles dépendances de la sortie d'un module vers son entrée. Considérons par exemple le morceau de système en figure 6.4.

Une solution consisterait à supposer que les dépendances de la sortie vers l'entrée doivent toutes avoir une composante non nulle dans l'extension de dimension. Ceci permettrait de « déstructurer » les modules de la même façon que nous avons déstructuré les variables, en considérant qu'il y a un module pour chaque point de l'extension de dimension. Outre le fait, somme toute peu gênant, que ceci conduirait éventuellement à gérer un nombre infini de modules, cette solution ne résout pas le problème des modules ayant des domaines d'entrée (formels, c'est-à-dire dès leur définition) infinis.

De la même façon que nous l'avons fait pour la sémantique opérationnelle *à la demande* dans le chapitre 3, nous allons donc devoir découper nos modules — et nos systèmes, qui doivent être




```
use { t | t >= 0 } cell (A,B) returns (C,D);
B[t]= case
    { | t=0 } : 0;
    { | t>0 } : D[t-1];
          esac;
```

FIG. 6.4 – *Un module rebouclé*

traités de la même façon pour assurer un minimum de compositionnalité — en « tranches » de composantes. Ces ensembles de composantes devront être clos pour la relation de dépendance, et la sémantique opérationnelle que nous définirons sera « paramétrée » par ceux-ci.

Nous proposons donc la modification suivante de la sémantique opérationnelle.

Notations Dans la suite, nous considérons que V est un sous-ensemble de Out, et nous noterons $W = \mathit{Nécessaire}(V)$. L'ensemble W est donc clos à gauche pour la relation de dépendance de S (au sens du chapitre 5).

Nous noterons Σ un état de S, σ un état de S' et Γ un état de $\overline{S}$.

Nous allons définir des transitions notées $\underset{V\ S}{\Longrightarrow}$ qui dérivent d'une légère modification des transitions *à la demande* que nous avions définies pour le sous-ensemble des composantes de sortie.

La différence avec la définition précédente tient dans la définition des états initiaux : un état initial est maintenant défini par rapport à W, comme l'ensemble des états valuant toutes les composantes de $W \cap In$, et aucune de $W \setminus In$

$$\mathit{Init}_S(W) = \{\Sigma \mid \forall x \in W \cap In : \Sigma(x) \neq \bot \wedge \forall x \in W \setminus In : \Sigma(x) = \bot\}$$

La définition d'un état final reste la même, avec juste une harmonisation des notations : un état final pour W est un état final pour $V = W \cap Out$ au sens donné au chapitre 3.

$$\mathit{Fin}_S(W) = \{\Sigma \mid V \subseteq \mathit{supp}(\Sigma)\}$$

Transitions « grands-pas » Nous devons maintenant définir des transitions pour S, en fonction des transitions pour S'. De plus, la sémantique obtenue devra être dans un certain sens équivalente à la sémantique du système substitué $\overline{S}$. Nous distinguons donc entre des transitions « petits-pas », qui sont les transitions classiques, *à la demande*, du système substitué, et des transitions « grands-pas », qui vont regrouper une série de transitions du module nécessaires à l'évaluation d'une composante de sortie de celui-ci. Le schéma en figure 6.5 expose cette l'idée : une série de transitions du module est remplacée par une seule transition de S.

Une transition de S est donc définie de la façon suivante

$$\Sigma \underset{V\ S}{\overset{x}{\Longrightarrow}} \Sigma'$$

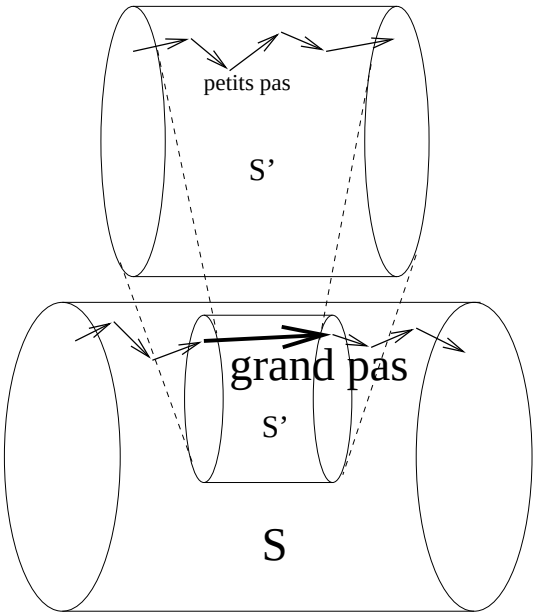


FIG. 6.5 – *Transitions « grands-pas »*

- si $x \notin Out'$, $\Sigma \Rightarrow \Sigma'$ est une transition à la demande classique ;
- si $x \in Out'$ tel que $\Sigma(x) = \bot$, on définit $V' = V \cap Out'$, $W' = N\acute{e}cessaire(V')$, et s'il existe $\sigma_0 \in Init_{S'}(W')$ avec

$$\sigma_0 \underset{W' \cap In'}{\sim} \Sigma$$

tel que

$$\sigma_0 \underset{V'}{\overset{*}{\Longrightarrow}}_{S'} \sigma$$

et

$$\sigma(x) \neq \bot$$

alors la transition $\Sigma \Rightarrow \Sigma'$ est définie avec

$$\Sigma' = \Sigma[x \leftarrow \sigma(x)]$$

Les états de S sont donc enrichis avec les valeurs des composantes, soit par calcul direct, soit par une suite de calculs dans le module. Ces dernières transitions sont *internes* au module, dans le sens où elles sont ensuite « oubliées » : si une autre composante de Out' nécessite le calcul des mêmes composantes locales au module, celui-ci devra être effectué à nouveau. Si nous considérons d'autre part le système substitué, ce recalcul n'a pas lieu : les composantes locales de S sont des composantes à part entière de $\overline{S}$. Vu du système appelant, le module se comporte comme une « boîte noire », ce qui est indispensable au caractère compositionnel de la sémantique[1] Nous devons donc maintenant montrer que cette sémantique grands-pas respecte bien le principe de substitution.

Considérons un système S, un ensemble V de ses composantes de sortie, et $W = N\acute{e}cessaire(V)$. Nous supposons que S a un module S', et que $\overline{S}$ est le système substitué correspondant. Nous

1. Nous n'élaborerons pas ici une preuve complète de compositionnalité, qui nécessiterait une généralisation à des systèmes comprenant des modules imbriqués.

posons[2]

$$\overline{V} = V$$

Nous montrons alors la proposition suivante

Proposition 19 (Équivalence des grands pas avec la substitution) *Si Σ et Γ sont des états initiaux respectivement de S et de $\overline{S}$ équivalents sur $W \cap In$, et si*

$$\Sigma \underset{V}{\overset{*}{\Longrightarrow}}_{S} \Sigma' \text{ avec } V \subseteq supp(\Sigma')$$

alors

$$\Gamma \underset{V}{\overset{*}{\Longrightarrow}}_{\overline{S}} \Gamma' \quad avec \quad V \subseteq supp(\Gamma') \quad et \quad \Gamma' \underset{W}{\sim} \Sigma'$$

et réciproquement.

Preuve. *La preuve de cette proposition est assez complexe, puisqu'il faut « reconstituer » et réarranger des suites de transitions en s'assurant de leur correction. Nous donnons l'essentiel de cette preuve en annexe.* ❑

6.4 Vers des preuves modulaires

La méthodologie de preuve que nous avons proposée au chapitre précédent reposait en grande partie sur la notion d'invariant canonique, qui permet de traduire toute la sémantique d'un système dans un langage logique externe. Nous ne pouvons malheureusement pas étendre directement cette approche aux programmes modulaires, pour la simple raison que nous n'avons pas accès de l'extérieur aux composantes locales au module. Il n'est pas possible de décrire les calculs internes au module — qui, dans la sémantique opérationnelle, sont masqués par les « grands pas » — et nous devons donc nous contenter d'un invariant plus faible.

Une solution possible est de reprendre la notion d'assertion univoque. En effet, celle-ci ne tient compte que des valeurs de sortie, et non de la façon dont celles-ci sont calculées. Nous redonnons ici la définition d'un invariant univoque, et montrons un premier résultat de modularité de cette notion.

Définition 30 (Invariant univoque) *Soit S un programme* ALPHA, *V un sous-ensemble de ses composantes de sortie, $W = Nécessaire_S(V)$. Une formule I sur les variables de S est un* invariant univoque *de S par rapport à V si*

- $\forall \Sigma_0 \in Init_S(W), \forall \Sigma_0 \underset{W}{\overset{*}{\Longrightarrow}}_{S} \Sigma$, *on a* $\Sigma \models I$
- *I est univoque par rapport à V au sens de la définition du chapitre 5.*

Cette définition suppose implicitement que la précondition est *true*. Il est possible de la généraliser à une précondition P quelconque en restreignant les états initiaux Σ_0 à ceux validant P.

Considérons maintenant un système S avec son module S, tels qu'ils soient tous deux *finis clos à gauche*, au sens du chapitre précédent. Considérons les ensembles V et W habituels, avec la condition que V soit fini. Posons

$$\begin{aligned} V' &= W \cap Out' \\ W' &= Nécessaire_{S'}(V') \end{aligned}$$

2. Rappelons que nous avons supposé les ensembles Out et Out' disjoints.

Les ensembles W, V' et W' sont donc également finis. Supposons maintenant que nous avons un invariant I' univoque de S' par rapport à V', qui ne met en jeu que des variables d'entrée ou de sortie de S'. Soit de plus I_C l'invariant canonique relatif aux équations de S (qui ne fait donc pas intervenir le module). Alors la conjonction $I_C \wedge I'$ permet de décrire le système S :

Proposition 20 (Modularité de l'invariant univoque) *La conjonction $I_C \wedge I'$ de l'invariant canonique de S et d'un invariant univoque de son module est un invariant univoque de S par rapport à V.*

Preuve. *Nous donnons l'essentiel de la preuve, semblable à celle de la proposition 17 du chapitre précédent. La preuve se fait par contraposée : supposons que la conjonction des deux invariants n'est pas univoque. Alors il existe deux états Σ et Σ' tels que*

$$\begin{array}{l} \Sigma \models I' \wedge I_C \\ \Sigma' \models I' \wedge I_C \\ V \subseteq supp(\Sigma) \\ V \subseteq supp(\Sigma') \\ \Sigma \underset{In}{\sim} \Sigma' \end{array} \qquad et \qquad \Sigma \not\underset{V}{\sim} \Sigma'$$

Il existe donc une composante x de V telle que $\Sigma(x) \neq \bot$, $\Sigma'(x) \neq \bot$ et $\Sigma(x) \neq \Sigma'(x)$. Considérons le cas plus général où $x \in W$. Deux cas apparaissent :

- *si x est défini par une équation, comme Σ et Σ' valident I_C, l'argument est le même que pour la preuve de la proposition 17 : il existe $y \preceq x$ tel que $\Sigma(y) \neq \Sigma'(y)$.*
- *si x est une composante de sortie du module. Considérons des états σ et σ' de S' coïncidant avec Σ et Σ' sur $In' \cup Out'$. Alors σ et σ' valident tous deux I'. De plus V' est inclus dans le support de σ et σ'. Par définition de l'invariant univoque appliquée au sous-système, on en déduit qu'il existe une composante y de $W' \cap In'$ telle que $\sigma(y) \neq \sigma'(y)$, donc $\Sigma(y) \neq \Sigma'(y)$.*

La suite de la preuve (argument de récurrence sur le chemin de dépendances fini) est identique à celle de la proposition 17. Il reste ensuite à montrer que $I_C \wedge I'$ est bien un invariant. Ceci se fait de la même façon que pour montrer que l'invariant canonique est un invariant. □

6.5 Discussion

Nous avons esquissé dans ce chapitre un traitement possible des programmes ALPHA structurés. La principale difficulté vient du fait qu'il faut définir une sémantique qui soit elle-même modulaire, voire compositionnelle. Le problème ici est que l'utilisation d'un module ne peut être vue comme un appel de procédure. On ne peut donc « attendre » que toutes les composantes d'entrée soient valuées avant d'effectuer les calculs internes au module. Une solution aurait été de restreindre un peu les possibilités de la structuration, en supposant que les domaines infinis ne peuvent apparaître qu'au plus haut-niveau. Dans ce cas, il aurait été possible de supposer toutes les entrées d'un module valuées, même si cela ne correspond pas au comportement opérationnel du système substitué. L'équivalence fonctionnelle qu'il est ensuite possible de prouver — et qui aurait été beaucoup plus simple à établir que celle donnée ici — aurait garanti la validité des preuves, puisque celles-ci concernent des propriétés fonctionnelles. Nous ne savons pas exactement à quel point cette restriction peut s'avérer contraignante. Nous avons donc exposé ici une solution plus générale mais

aussi plus complexe, qui consiste à paramétrer la sémantique opérationnelle par des ensembles de composantes.

En ce qui concerne les preuves par invariants, la notion d'invariant univoque semble être la plus utile, puisque c'est cet invariant qui est le plus « fort » si l'on ne tient pas compte des composantes locales aux modules. Contrairement au cas de l'invariant canonique, nous n'avons pas de méthode « automatique » pour trouver un invariant univoque d'un système. Dans le cas d'un système ne faisant pas de calcul inductif, celui-ci peut être déduit d'un nombre fini de substitutions. Dans le cas contraire, le problème est bien plus difficile, puisqu'il faut faire appel à l'intuition pour obtenir la version non inductive du calcul. L'exemple de calcul de la fonction de temps du chapitre précédent est là pour nous en persuader. L'« automatisation » totale d'une preuve modulaire ne semble donc pas possible, mais

- la notion même de preuve modulaire laisse à penser que les spécifications des modules sont données par l'utilisateur, qui sait donc quelles informations sont nécessaires pour établir les propriétés voulues ;
- ces « informations nécessaires » ne sont pas obligatoirement aussi riches que celles données par un invariant univoque, et donc plus simples à générer. Des propriétés plus simples doivent être exprimées par des invariants plus simples.

chapitre 7

Conclusion

Nous avons proposé dans cette seconde partie un cadre logique pour la validation de programmes ALPHA. Celui-ci est inspiré d'une méthode « à la Hoare », qui consiste à écrire des spécifications où le programme est équipé d'une précondition et d'une postcondition. Cette méthode repose sur la définition d'une sémantique opérationnelle, qui permet de raisonner sur des états initiaux et finaux, et sur les transitions entre ces états. L'approche proposée est une approche « tout logique », qui « oublie » la structure syntaxique des programmes ALPHA.

La méthodologie de preuve utilise une notion d'invariant. Deux types d'invariants particuliers se dégagent : les invariants canoniques, qui correspondent à une description opérationnelle du programme, et les invariants univoques, qui correspondent plutôt à une description dénotationnelle. Les preuves se font au niveau logique, principalement en prouvant des implications entre les invariants.

Nous avons dans le dernier chapitre proposé une piste pour le traitement des programmes structurés par l'ajout de modules. Ce dernier point est plus difficile à traiter, à cause de la présence simultanée d'un nombre potentiellement infini de composantes et d'un mécanisme d'évaluation de type « paresseux » des entrées des modules.

L'approche opérationnelle que nous avons choisie, outre le fait qu'elle fournissait une modélisation utilisée ensuite dans le cadre de preuve, correspondait à un besoin de comprendre comment il est possible d'« exécuter » un programme ALPHA. Nous n'avons pas formellement établi le lien avec la sémantique par point fixe de Mauras. Ce point aurait sans doute pu être traité plus en détail. Il serait par ailleurs probablement possible, plutôt que de conserver une approche « tout logique », d'adopter une vision mixte, plus orientée dans le sens de Floyd que dans celui de Hoare. Une telle vision tirerait parti à la fois de la richesse supplémentaire apportée par la logique, et des nombreux résultats sur le modèle polyédrique qui servent de base au système ALPHA.

Ce dernier point nous amène tout naturellement à envisager la suite du travail. Outre un approfondissement de la réflexion sur les preuves modulaires, celle-ci consiste principalement en une intégration d'un outil automatique de preuve dans l'environnement ALPHA. Cet outil doit permettre une interaction entre l'environnement ALPHA proprement dit, qui manipule les programmes au niveau syntaxique, et un autre outil (du type prouveur de théorèmes) qui traiterait les aspects logiques. L'originalité de la démarche proposée tient dans le fait qu'elle permet alors de créer un lien fort entre ALPHA et les nombreux travaux effectués dans le domaine de la preuve formelle « directe » de circuits.

Conclusion

Cette thèse s'inscrit dans le cadre des recherches sur les langages à parallélisme de données. La première partie, qui concerne le langage $\mathcal{L}$, prolonge et effectue une synthèse des travaux commencés avec la définition du langage au début de cette décennie. Elle montre qu'il est possible de tirer parti de la relative simplicité du modèle de programmation data-parallèle pour établir un cadre formel à la fois précis et utilisable d'un point de vue pratique. La position du langage $\mathcal{L}$, à l'« intersection » de plusieurs langages impératifs existants, permet, à condition de compléter un peu les fonctionnalités du langage, d'étendre les résultats obtenus à ces différents langages « réels ». Le parallélisme de données paraît donc de ce point de vue un modèle « solide » pour la programmation massivement parallèle. Le langage ALPHA est à la fois moins général et plus tourné vers des applications directes. Vu initialement comme un moyen de dériver automatiquement des circuits appartenant à une certaine classe, il est apparu ensuite comme un moyen plus général de spécifier des calculs data-parallèles. Dans chacune de ces deux directions, un besoin de validation s'est fait sentir, notamment avec l'accroissement de la taille des applications envisagées. Le travail décrit ici, s'appuyant sur le fait qu'ALPHA est un *langage* permettant de décrire des *circuits*, doit permettre à terme de faire le lien entre preuve de programmes et preuves de circuits, deux domaines qui ont tendance à s'ignorer.

Reprenons brièvement les grandes lignes du travail présenté dans cette thèse.

- En ce qui concerne le langage $\mathcal{L}$, nous avons poursuivi les travaux de Bougé & al. sur un premier système de preuve et le calcul des plus faibles préconditions associé au premier langage d'assertion. Le but était de définir une méthode de preuve à la fois solide du point de vue théorique, et qui se rapprochait le plus possible d'une méthode scalaire classique. Au niveau théorique, nous avons montré que, malgré certains problèmes d'expressivité, le langage des assertions en deux parties possédait des propriétés de complétude suffisantes pour envisager une utilisation pratique. Ces résultats de complétude partielle, antérieurs à ceux de Utard qui ont nécessité une modification du calcul des plus faibles préconditions, nous ont en effet permis de définir une méthode de preuve par annotations. Cette méthode est identique à celle que l'on utilise habituellement pour les langages impératifs scalaires, si ce n'est qu'elle requiert une première phase d'étiquetage syntaxique. C'est probablement la contribution la plus intéressante au niveau de ce langage des assertions en deux parties : montrer qu'il est effectivement utilisable de façon simple. Cette méthode serait de plus aisément automatisable, par la génération de *conditions de vérification* à la Gordon. Un premier prototype, incomplet car sans preuve en deux phases, avait d'ailleurs été développé par Mounier et Utard ([53]).

- Le langage en deux parties souffrait cependant de problèmes au niveau théorique. Ceux-ci sont apparus dans la première preuve de complétude que nous avons donnée, qui est restée limitée aux programmes sans boucles. La modification du calcul des plus faibles préconditions définie par Utard suggérait de recourir à une autre approche. C'est l'idée du second langage d'assertion que nous avons défini ensemble. Plus « élégant », celui-ci est en revanche moins

pratique au niveau de la preuve par annotations. Une troisième voie, qui place les substitutions au niveau d'une transformation syntaxique, semble montrer que ces substitutions sont inévitables, et qu'elles sont le prix à payer pour pouvoir manipuler un contexte d'activité dans les preuves.

- La seconde partie, traitant du langage ALPHA, a nécessité un travail de défrichement plus important. Le besoin d'un cadre externe, pour exprimer les propriétés voulues tout en conservant au langage son efficacité, écartait l'utilisation de méthodes de validation liées aux langages fonctionnels. Il nous a paru intéressant en revanche d'adapter l'idée de la logique de Hoare à ce cas précis. Il nous a fallu procéder à des travaux préliminaires, afin de disposer d'une base de travail : c'est le but de la définition d'une sémantique opérationnelle, qui formalise le comportement des programmes. Nous avons ensuite montré qu'il était possible de passer du « monde » ALPHA au « monde » de la logique, et ainsi établir de façon formelle des propriétés fonctionnelles de systèmes ALPHA. Nous avons finalement esquissé une réponse au problème de la structuration des preuves. L'ensemble de ces résultats fournit un cadre original pour la validation de programmes ALPHA.

Le travail sur le langage $\mathcal{L}$ a atteint un niveau de maturité que l'on peut juger satisfaisant, dans le sens où le travail qui reste à faire consiste à l'appliquer à des « langages-cibles » réels. Des extensions aux structures d'échappement ont déjà été proposées, qui permettent de « simuler » des caractéristiques plus complexes. Un autre complément en ce sens est apporté dans cette thèse par l'introduction de l'instruction d'affectation inconditionnelle. L'intérêt du travail effectué ici est donc d'avoir posé des bases solides pour l'implantation d'un outil de validation qui concernerait un langage « réel ». Un point cependant resterait à traiter : celui de l'extension à des instructions (ou directives) de placement des données.

Le travail sur ALPHA nécessite un approfondissement plus immédiat. Les perspectives à court terme sont de deux types. Il faut d'une part affiner et prolonger les résultats afin de considérer plusieurs extensions :

- Améliorer et prolonger les travaux sur la modularité. Qu'attend-on concrètement d'une preuve modulaire ? Qu'est-il raisonnable d'espérer ?

- Envisager la preuve de propriétés non fonctionnelles, par exemple des propriétés de sûreté ou de vivacité, ou au « bas niveau » des propriétés portant sur le contrôle des circuits.

D'autre part, il reste à intégrer un outil de preuve dans l'outil ALPHA. Cet outil s'appuierait naturellement sur le cadre logique proposé, mais probablement davantage sur une approche semi-logique qui permettrait de tirer parti à la fois de la logique externe et des transformations syntaxiques. Cette intégration permet d'ouvrir de nouvelles perspectives pour le langage ALPHA, dans le sens où elle fait le lien entre celui-ci et le vaste domaine de la preuve de circuits. Une première tentative d'implantation est en cours avec le système PVS [56, 19], utilisé à de nombreuses reprises pour prouver des circuits. Cette démarche doit permettre à terme de profiter des apports des deux « communautés » : disposer d'un langage de spécification de haut niveau qui permet d'automatiser le travail de traduction vers le circuit d'une part, la logique d'autre part, et prouver que les implantations réalisées sont effectivement correctes.

Bibliographie

[1] Apt (K. R.) et Olderog (E.-R.). – *Verification of Sequential and Concurrent Programs.* – Springer Verlag, 1991, *Text and Monographs in Computer Science.*

[2] Bougé (L.). – *On the semantics of languages for massively parallel architectures.* – Rapport de recherche n° 90–06, ENS Ulm, Paris, LIENS, 1990.

[3] Bougé (L.). – The data-parallel programming model: A semantic perspective. *In: The data-parallel programming model*, éd. par Perrin (G.-R.) et Darte (A.), pp. 4–26. – Springer Verlag, 1996.

[4] Bougé (L.), Cachera (D.), Le Guyadec (Y.), Utard (G.) et Virot (B.). – Formal validation of data-parallel programs: introducing the assertional approach. *In: The data-parallel programming model*, éd. par Perrin (G.-R.) et Darte (A.), pp. 252–281. – Springer Verlag, 1996.

[5] Bougé (L.), Cachera (D.), Le Guyadec (Y.), Utard (G.) et Virot (B.). – Formal validation of data-parallel programs: a two-component assertional proof system for a simple language. *Theoretical Computer Science B*, 1997. – À paraître.

[6] Bougé (L.) et Utard (G.). – *Escape constructs in data-parallel languages: semantics and proof system.* – Rapport de recherche n° 94–18, ENS Lyon, LIP, 1994.
Url: `ftp://ftp.lip.ens-lyon.fr/pub/Rapports/RR/RR94-18.ps.Z`.

[7] Bougé (L.) et Cachera (D.). – On the completeness of a proof system for a simple data-parallel language. *In: Proc. 1st EuroPar Conf.* – Stockholm, Suède, août 1995. LNCS 966.

[8] Bougé (L.) et Cachera (D.). – Proving data-parallel programs correct: The proof outlines approach. *In: Proc. Massively Parallel Programming Models.* – Berlin, Allemagne, octobre 1995. IEEE.

[9] Bougé (L.) et Cachera (D.). – A logical framework to prove properties of Alpha programs. *In: Proc. Application-Specific Arrays and Processors*, pp. 187–198. – Zürich, Suisse, juillet 1997. IEEE.

[10] Bougé (L.), Le Guyadec (Y.), Utard (G.) et Virot (B.). – On the expressivity of a weakest preconditions calculus for a simple data-parallel programming language. *In: ConPar–VAPP: joint international conference on vector and parallel processing.* – Linz, Austria, septembre 1994. LNCS 854.

[11] Bougé (L.), Le Guyadec (Y.), Utard (G.) et Virot (B.). – A proof system for a simple data-parallel programming language. *In: Proc. of IFIP WG 10.3, Applications in Parallel and Distributed Computing*, éd. par Girault (C.). – Caracas, Venezuela, avril 1994.

[12] Bougé (L.) et Levaire (J.-L.). – Control structures for data-parallel SIMD languages: semantics and implementation. *Future Generation Computer Systems*, vol. 8, 1992, pp. 363–378.

[13] Cachera (D.). – *Étude de systèmes de preuve pour les programmes data-parallèles.* – Rapport de DEA, ENS Lyon, France, 1994.

[14] Cachera (D.). – Deux résultats de complétude pour un langage data-parallèle simple. *In: Proc. RenPar'7*, éd. par Dekeyser, Libert et Manneback. – Mons, Belgique, mai 1995.

[15] Cachera (D.). – Prouver des programmes Alpha: l'approche externe. *In: Proc. RenPar'9*, éd. par Schiper (A.) et Trystram (D.). – École Polytechnique Fédérale de Lausanne, Suisse, mai 1997.

[16] Cachera (D.) et Utard (G.). – Proving data-parallel programs: a unifying approach. *Parallel Processing Letters*, vol. 6, n° 4, 1996, pp. 491–505.

[17] Caspi (P.), Halbwachs (N.), Pilaud (D.) et Plaice (J. A.). – Lustre: a declarative language for programming synchronous systems. *In: Proc. 14th Annual ACM Symposium on Principles of Programming Languages*, pp. 178–188. – München, Allemagne, 1987.

[18] Chandy (K.M) et Misra (J.). – *Parallel program design: a foundation.* – Addison-Wesley, 1988.

[19] Chevalier (Y.). – *Modules en Alpha. Définition, sémantique, équivalence.* – Rapport de DEA, École normale supérieure de Lyon, France, juin 1997.

[20] Clint (M.) et Narayana (K. T.). – On the completeness of a proof system for a synchronous parallel programming langage. *In: Third Conf. Found. Softw. Techn. and Theor. Comp. Science.* – Bangalore, India, décembre 1983.

[21] Clint (M.) et Narayana (K. T.). – Programming structures for synchronous parallelism. *In: Int. Conf. "Parallel Computing 83"*, éd. par B.V. (Elsevier Science Publishers). – North-Holland, 1984.

[22] Cook (S. A.). – Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computation*, vol. 7, n° 1, février 1978, pp. 70–90.

[23] Cousot (P.). – Methods and logics for proving programs. *In: Handbook of Theoretical Computer Science*, éd. par Van Leeuwen (J.), chap. 15. – Elsevier Science, 1990.

[24] Darte (A.). – *Techniques de parallélisation automatique de nids de boucles.* – Thèse de doctorat, École normale supérieure de Lyon, France, 1993.

[25] De Bakker (J.). – *Mathematical Theory of Program Correctness.* – Prentice-Hall, 1981.

[26] Dezan (C.) et Quinton (P.). – *Verification of regular architectures using Alpha: a case study.* – Rapport de recherche n° 823, Rennes, France, IRISA, 1994.

[27] Dijkstra (E. W.). – *A Discipline of Programming.* – Prentice-Hall, 1976.

[28] Dion (M.). – *Alignement et distribution en parallélisation automatique.* – Thèse de doctorat, École normale supérieure de Lyon, France, 1996.

[29] DPCE Subcommittee Numerical C Extensions Group of X3J11. – *Data-Parallel C Extension.* – ANSI, décembre 1994.

[30] Dupont de Dinechin (F.). – *Systèmes structurés d'équations récurrentes : mise en œuvre dans le langage Alpha et applications.* – Thèse de doctorat, Univ. Rennes I, France, janvier 1997.

[31] Floyd (R.). – Assigning meaning to programs. *In : Proc. Symposium on Applied Mathematics 19, Mathematical Aspects of Computer Science*, éd. par Schwartz (J. T.). – American Mathematical Society, New York, 1967.

[32] Flynn (M. J.). – Some computer organizations and their effectiveness. *IEEE Transactions on computers*, vol. 21, n° 9, 1972, pp. 948–960.

[33] Gabarró (J.) et Gavaldà (R.). – An approach to correctness of data-parallel algorithms. *Journal of Parallel and Distributed Computing*, vol. 22, n° 2, août 1994, pp. 185–201.

[34] Girault (A.). – *Sur la Répartition de Programmes Synchrones.* – Thèse de doctorat, INPG, Grenoble, France, janvier 1994.

[35] Gordon (M.). – Why higher order logic is a good formalism specifying and verifying hardware. *In : Formal aspects of VLSI design, Proc. of the 1985 Edinburgh conference on VLSI*, éd. par Milne (J.) et Subrahmanyam (P. A.). – North-Holland, 1986.

[36] Gordon (M. J. C.). – *Programming Language Theory and its Implementation.* – Prentice-Hall International, 1988.

[37] Gries (D.). – *The Science of Programming.* – Springer Verlag, 1981.

[38] Hatcher (P. J.) et Quinn (M. J.). – *Data-Parallel Programming on MIMD Computers.* – MIT Press, 1991.

[39] Hillis (W. D.) et Steele (G. L.). – Data parallel algorithms. *Comm. ACM*, vol. 29, n° 12, 1986, pp. 1170–1183.

[40] Hoare (C. A. R.). – An axiomatic basis for computer programming. *Comm. ACM*, vol. 12, n° 10, 1969, pp. 576–583.

[41] Hoare (C. A. R.). – Communicating sequential processes. *Comm. ACM*, vol. 21, 1978, pp. 666–677.

[42] Jones (G.) et Sheeran (M.). – Designing arithmetic circuits by refinement in Ruby. *In : Mathematics of Program Construction*, éd. par Bird (R. S.), Morgan (C. C.) et Woodcock (J. C. P.). – Springer Verlag, 1993.

[43] Karp (R. M.), Miller (R. E.) et Winograd (S.). – The organization of computations for uniform recurrence equations. *Journal of the ACM*, vol. 14, n° 3, juillet 1967, pp. 563–590.

[44] Koelbel (C. H.), Loveman (D. B.), Schreiber (R. S.), Steele (G.L.) et Zosel (M. E.). – *The High Performance Fortran handbook.* – MIT Press, 1994.

[45] Lamport (L.). – Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, vol. 10, n° 2, avril 1988, pp. 267–281.

[46] Le Guyadec (Y.) et Virot (B.). – *Axiomatics semantics of conditioning constructs and non-local transfers in data-parallel languages.* – Rapport de recherche, Université d'Orléans, France, LIFO, 1994.

[47] Le Guyadec (Y.) et Virot (B.). – Sequential-like proofs of data-parallel programs. *Parallel Processing Letters*, vol. 6, n° 3, 1996.

[48] Le Verge (H.). – *Un environnement de transformations de programmes pour la synthèse d'architectures régulières.* – Thèse de doctorat, Univ. Rennes I, Rennes, France, décembre 1989.

[49] Le Verge (H.), Mauras (C.) et Quinton (P.). – A language-oriented approach to the design of systolic chips. *Journal of VLSI Signal Processing*, vol. 3, 1991, pp. 173–182.

[50] Levaire (J.-L.). – *Contribution à l'étude sémantique des langages à parallélisme de données.* – Thèse de doctorat, Univ. Paris 7, LIP, ENS Lyon, 1993.

[51] MasPar Computer Corporation, Sunnyvale CA. – *Maspar Parallel Application Language Reference Manual*, 1990.

[52] Mauras (C.). – *Alpha: un langage équationnel pour la conception et la programmation d'architectures systoliques.* – Thèse de doctorat, Univ. Rennes I, France, décembre 1989.

[53] Mounier (L.) et Utard (G.). – *Sémantique axiomatique des langages à parallélisme de données; automatisation de la preuve de programmes.* – Rapport de recherche n° 93–08, ENS Lyon, LIP, 1993.

[54] Nédelka (L.). – *Étude expérimentale des systèmes de transitions des programmes Alpha.* – Rapport de DEA, IRISA, Rennes, France, 1995.

[55] Owicki (S.) et Gries (D.). – Verifying Properties of Parallel Programs: An Axiomatic Approach. *Comm. ACM*, vol. 19, n° 5, 1976, pp. 279–285.

[56] Owre (S.), Rushby (J. M.) et Shankar (N.). – *The PVS specification language.* – Rapport de recherche, Menlo Park, CA, SRI, juin 1993.

[57] Paris (N.). – *HyperC Specification Document.* – Rapport de recherche n° 93–1, École Polytechnique, Palaiseau, France, HyperParallel Technologies, 1993.

[58] Perrott (R. H.). – A language for array and vector processors. *ACM Transactions on Programming Languages and Systems*, vol. 1, n° 2, octobre 1979, pp. 294–312.

[59] Purushothaman (S.) et Subrahmanyam (P. A.). – Mechanical certification of systolic algorithms. *Journal of Automated Reasoning*, vol. 5, n° 1, 1989, pp. 67–95.

[60] Quinton (P.), Rajopadhye (S. V.) et Risset (T.). – Extension of the alpha language to recurrences on sparse periodic domains. *In: Application Specific Array Processors.* – Chicago, USA, 1996. IEEE.

[61] Quinton (P.), Rajopadhye (S. V.) et Wilde (D.). – *Deriving data-parallel code from functional programs.* – Rapport de recherche n° 905, Rennes, France, IRISA, janvier 1995.

[62] Rajopadhye (S. V.). – An improved systolic algorithm for the algebraic path problem. *Integration: The VLSI Journal*, vol. 14, n° 3, février 1993, pp. 279–296.

[63] Saouter (Y.). – *À propos de systèmes d'équations récurrentes.* – Thèse de doctorat, Univ. Rennes I, France, octobre 1992.

[64] Stewart (A.). – Reasoning about data-parallel array assignment. *Journal of Parallel and Distributed Computing*, vol. 27, 1985, pp. 79–85.

[65] Stewart (A.). – SIMD language design using prescriptive semantics. *BIT*, vol. 28, 1988, pp. 639–650.

[66] Stewart (A.). – An axiomatic treatment of SIMD assignment. *BIT*, vol. 30, 1990, pp. 70–82.

[67] Thinking Machine Corporation, Cambridge MA. – *C* programming guide*, 1990.

[68] Utard (G.). – *Sémantique des langages à parallélisme de données. Applications à la validation et à la compilation.* – Thèse de doctorat, École normale supérieure de Lyon, 1995.

[69] Wilde (D. K.). – *The Alpha language.* – Rapport de recherche n° 999, Rennes, France, IRISA, janvier 1994.

[70] Winskel (G.). – *The Formal Semantics of Programming Languages: An introduction.* – The MIT Press, 1993.

[71] Wray (J. P.) et Stewart (A.). – *Complementary and Consistent Semantics for SIMD Computation.* – Rapport de recherche, Belfast, Great-Britain, The Queen's University, 1989.

Annexes

annexe **A**

Compléments sur $\mathcal{L}$

Nous donnons ici les preuves de quelques résultats concernant les systèmes de preuve pour le langage $\mathcal{L}$.

1.1 Complétude

Nous commençons avec des résultats de complétude du premier système de preuve pour $\mathcal{L}$, donnés au chapitre 4. Tout d'abord, nous prouvons la proposition concernant le conditionnement d'un programme non régulier. Rappelons les notations : nous avons un programme S, et une assertion $\{Q, D\}$ qui tient lieu de postcondition. Nous voulons montrer que, si nous pouvons exprimer la plus faible précondition de S avec une certaine postcondition dérivée de $\{Q, D\}$, alors la plus faible précondition de S pour $\{Q, D\}$ sera elle aussi exprimable par une assertion. Ceci est exprimé par la proposition suivante.

Proposition 4 *Soit $(S, \{Q, D\})$ une paire simple, et soit Tmp une variable telle que $Tmp \notin Var(Q) \cup Var(S) \cup Var(D)$. Si*

$$wp(S, \{Q, D \wedge Tmp\}) = \{P, C\}$$

alors

$$wp(\textsf{where } B \textsf{ do } S \textsf{ end}, \{Q, D\}) = \{P[B/Tmp], C\}$$

Preuve. *Supposons que $wp(S, \{Q, D \wedge Tmp\}) = \{P, C\}$.*

Soit (σ, c) dans $wp(\textsf{where } B \textsf{ do } S, \{Q, D\})$. D'après la sémantique du conditionnement,

$$[\![\textsf{where } B \textsf{ do } S]\!](\sigma, c) = (\sigma', c) \in [\![\{Q, D\}]\!],$$

avec σ' tel que $(\sigma', c \wedge \sigma(B)) = [\![S]\!](\sigma, c \wedge \sigma(B))$.

Nous avons $\sigma' \models Q$ et $\sigma'(D) = c$, et $Var(D) \cap Change(S) = \emptyset$ implique $\sigma'(D) = \sigma(D) = c$.

Posons $\sigma_1 = \sigma[Tmp \leftarrow \sigma(B)]$ et $\sigma'_1 = \sigma'[Tmp \leftarrow \sigma(B)]$. Par définition, $\sigma_1(Tmp) = \sigma(Tmp) = \sigma(B)$, et comme $Tmp \notin Var(S)$, $\sigma_1(D) = \sigma(D) = c = \sigma'(D) = \sigma'_1(D)$. De plus, $Tmp \notin Var(Q)$. Donc $\sigma'_1 \models Q$.

Comme $Tmp \notin Var(S)$, nous avons $[\![S]\!](\sigma_1, c \wedge \sigma(B)) = (\sigma'_1, c \wedge \sigma(B)) = (\sigma'_1, \sigma'_1(D \wedge Tmp))$.

Mais $(\sigma'_1, \sigma'_1(D \wedge Tmp)) \models \{Q, D \wedge Tmp\}$. Donc $(\sigma_1, c \wedge \sigma(B)) \models \{P, C\}$. En particulier, nous avons $\sigma_1 \models P$. Par le lemme de substitution, $\sigma \models P[B/Tmp]$, et

$$(\sigma, c) \models \{P[B/Tmp], D\}.$$

Réciproquement, soit (σ, c) *dans* $[\![\{P[B/Tmp], D\}]\!]$. *Posons* $\sigma_1 = \sigma[Tmp \leftarrow \sigma(B)]$. *Par le lemme de substitution,* $\sigma_1 \models P$. *De plus, comme* $Tmp \notin Var(D)$, $\sigma_1(D) = \sigma(D) = c$, *donc* $(\sigma_1, c \wedge \sigma_1(Tmp)) \models \{P, D \wedge Tmp\}$.

Par hypothèse et par le lemme d'extension, $wp(S, \{Q, D \wedge Tmp\}) = \{P, D \wedge Tmp\}$. *Soit* $[\![S]\!](\sigma_1, c \wedge \sigma_1(Tmp) = (\sigma'_1, c \wedge \sigma_1(Tmp))$.

Nous avons $(\sigma'_1, c \wedge \sigma_1(Tmp)) \models \{Q, D \wedge Tmp\}$, *et en particulier* $\sigma'_1 \models Q$.

Soit maintenant $\sigma' = \sigma'_1[Tmp \leftarrow \sigma(Tmp)]$. *Comme* $Tmp \notin Var(S)$, $[\![S]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c \wedge \sigma(B))$. *Mais* $\sigma'_1 \models Q$ *and* $Tmp \notin Var(Q)$, *donc* $\sigma' \models Q$.

Finalement, $Tmp \notin Var(D)$, *donc* $\sigma'(D) = c$. *Nous avons donc*

$$[\![\textsf{where } B \textsf{ do } S]\!](\sigma, c) = (\sigma', c) \in [\![\{Q, D\}]\!].$$

□

Nous donnons maintenant la preuve du théorème de complétude pour les spécifications simples, dans le cas linéaire et régulier.

Théorème 2 (Complétude, cas simple, linéaire, régulier) *Soit* $\{P, C\}\ S\ \{Q, D\}$ *une spécification simple, où* S *est un programme linéaire et régulier. Si*

$$\models \{P, C\}\ S\ \{Q, D\}$$

alors

$$\vdash \{P, C\}\ S\ \{Q, D\}.$$

Preuve. *La preuve de ce théorème suit le schéma donné dans [1]. Elle utilise le calcul des plus faibles préconditions. Pour tout programme linéaire et régulier* S, *et pour toute spécification simple* $\{\ldots\}\ S\ \{Q, D\}$, *il existe une assertion* $\{P', C'\}$ *telle que* $[\![\{P', C'\}]\!] = wp(S, \{Q, D\})$. *En utilisant la règle de conséquence, il suffit donc de montrer que* $\vdash \{wp(S, \{Q, D\})\}\ S\ \{Q, D\}$.

La preuve se fait par induction sur la structure de S, *en utilisant les propriétés de définissabilité du théorème 1.*

Les cas de l'affectation et de séquencement sont immédiats. Considérons le cas du conditionnement, avec $S \equiv$ where B do T. *Comme* S *est régulier par hypothèse, nous avons* $Change(T) \cap Var(B) = \emptyset$. *Comme la spécification est simple, nous avons* $Change(S) \cap Var(D) = \emptyset$. *Comme* $Change(T) = Change(S)$, *nous avons aussi* $Change(T) \cap Var(D) = \emptyset$. *La propriété de définissabilité donne alors une assertion* $\{P, C\}$ *telle que* $\{P, C\} = wp(T, \{Q, D \wedge B\})$. *Par le lemme d'extension, nous obtenons* $wp(T, \{Q, D \wedge B\}) = \{P, D \wedge B\}$.

Le programme T *est régulier et linéaire, puisque* S *l'est. La spécification* $\{P, D \wedge B\}\ T\ \{Q, D \wedge B\}$ *est simple, donc l'hypothèse d'induction donne*

$$\vdash \{P, D \wedge B\}\ T\ \{Q, D \wedge B\}$$

Comme $(Var(B) \cup Var(D)) \cap Change(T) = \emptyset$, *la règle du* where *du système de preuve s'applique et nous obtenons*

$$\vdash \{P, D\}\ \textsf{where } B \textsf{ do } T\ \{Q, D\}$$

De plus, la propriété de définissabilité donne $wp($where B do $T, \{Q, D\}) = \{P, D\}$. *D'où le résultat désiré :* $\vdash \{wp($where B do $T, \{Q, D\})\}$ where B do $T\ \{Q, D\}$.

Considérons maintenant une spécification simple avec $\models \{P, C\}\ S\ \{Q, D\}$, *où* S *est linéaire et régulier. Par la propriété de définissabilité, il existe une assertion* $\{P', C'\}$ *telle que* $\{P', C'\} = wp(S, \{Q, D\})$. *D'après le résultat précédent, nous savons que* $\vdash \{P', C'\}\ S\ \{Q, D\}$. *Par le lemme de conséquence 2, nous obtenons* $\{P, C\} \Rightarrow \{P', C'\}$. *Nous pouvons donc appliquer la règle de conséquence du système de preuve. Ceci donne* $\vdash \{P, C\}\ S\ \{Q, D\}$ *comme attendu.* ❑

1.2 Preuves par annotations

Nous donnons maintenant la preuve de l'équivalence entre preuves par annotations et preuves dans le système $\vdash^*$, énoncée au chapitre 5. La proposition à prouver est la suivante :

Proposition 7 *Soit* S *un sous-programme étiqueté par* i, *et* P *et* Q *deux assertions telles que* $\forall j > 0, Tmp_j \notin Var(Q)$. *Alors*

$$\{P\}\ S^*\ \{Q\}$$

est une annotation pour S *si et seulement si*

$$\vdash^* \{P, \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}$$

Nous donnons d'abord la preuve du sens direct : s'il existe une preuve par annotations, alors la spécification est dérivable dans $\vdash^*$.

Preuve. *Soit* S *un sous-programme étiqueté par* i, *et* $\{P\}\ S^*\ \{Q\}$ *une preuve par annotations pour* S. *La preuve se fait par induction sur la longueur de la construction de la preuve par annotations. Nous avons cinq cas à considérer, correspondant respectivement à chaque règle de dérivation de la preuve par annotations.*

- *Si la dernière règle appliquée était*

$$\frac{\forall j > i, Tmp_j \notin Var(Q)}{\{Q[\bigwedge_{k=0}^{i} Tmp_k ? E : X / X]\}\ (i)\ \ \mathsf{X} := \mathsf{E}\ \{Q\}},$$

  alors, comme $X \notin \{Tmp_i \mid i \in \mathbb{N}\}$, *nous avons* $\vdash^* \{P, \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}$.

- *Si la dernière règle appliquée était*

$$\frac{P \Rightarrow P' \quad \{P'\}\ S^*\ \{Q'\} \quad Q' \Rightarrow Q \quad \forall j > Lab(S), Tmp_j \notin Var(Q) \cup Var(Q')}{\{P\}\{P'\}\ S^*\ \{Q'\}\{Q\}},$$

  alors par hypothèse d'induction nous avons

$$\vdash^* \{P', Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q', \bigwedge_{k=0}^{i} Tmp_i\},$$

  donc la règle de conséquence de $\vdash^*$ *s'applique et donne le résultat désiré.*

- *Si la dernière règle appliquée était la règle de composition séquentielle, alors il existe S_1 et S_2 tels que $S = S_1;S_2$, et une assertion R telle que nous ayons les preuves par annotations $\{P\}\ S_1^*\ \{R\}$ et $\{R\}\ S_2^*\ \{Q\}$. De plus, nous savons que S_1 et S_2 sont étiquetés par le même entier i. Par la règle de composition séquentielle dans les preuves par annotations, nous avons $\forall j > i, Tmp_j \notin Var\, R$. Par hypothèse d'induction, nous avons donc*

$$\vdash^* \{P, \bigwedge_{k=0}^{i} Tmp_i\}\ S_1^*\ \{R, \bigwedge_{k=0}^{i} Tmp_i\}$$

et

$$\vdash^* \{R, \bigwedge_{k=0}^{i} Tmp_i\}\ S_2^*\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}.$$

La règle de séquencement de $\vdash^*$ s'applique donc et donne

$$\vdash^* \{P, \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}.$$

- *Si la dernière règle appliquée était*

$$\frac{\{P'\}\ T^*\ \{Q\} \quad Lab(T) = i+1 \quad \forall j > i, Tmp_j \notin Var(Q)}{\{P'[B/Tmp_{i+1}]\}\ (i)\ \ \begin{array}{l}\mathsf{where}\ B\ \mathsf{do}\ \ [Tmp_{i+1} \equiv B] \\ \quad T^* \\ \mathsf{end}\ \{Q\}\end{array}}$$

avec $P = P'[B/Tmp_{i+1}]$. Nous avons $\forall j > i+1, Tmp_j \notin Var(Q)$, donc par hypothèse d'induction

$$\vdash^* \{P', \bigwedge_{k=0}^{i} Tmp_i \wedge Tmp_{i+1}\}\ T\ \{Q, \bigwedge_{k=0}^{i} Tmp_i \wedge Tmp_{i+1}\}$$

Comme $\{P' \wedge Tmp_{i+1} = B, \bigwedge_{k=0}^{i} Tmp_i \wedge B\} \Rightarrow \{P', \bigwedge_{k=0}^{i} Tmp_i \wedge Tmp_{i+1}\}$, la règle de conséquence donne

$$\vdash^* \{P' \wedge Tmp_{i+1} = B, \bigwedge_{k=0}^{i} Tmp_i \wedge B\}\ T\ \{Q, \bigwedge_{k=0}^{i} Tmp_i \wedge Tmp_{i+1}\}.$$

La règle du where *s'applique alors et donne*

$$\vdash^* \{P' \wedge Tmp_{i+1} = B, \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}.$$

Finalement, en utilisant la règle de substitution avec B/Tmp_{i+1}, nous avons

$$\vdash^* \{P, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}.$$

- *Le dernier cas (élimination d'assertions dans la preuve par annotations) est immédiat.*

Ceci termine la première partie de la preuve. ❑

Nous voulons maintenant prouver la réciproque. Nous allons utiliser le calcul des plus faibles préconditions, et aurons besoin des résultats préliminaires suivants.

Lemme A.1 *Soit Z une variable, S un programme, Q une assertion et D une expression booléenne telle que $Var(D) \cap Change(S) = \emptyset$. Si*

$$Z \notin Var(S) \cup Var(Q) \cup Var(D),$$

alors il existe une assertion $\{P, C\}$ telle que

$$wp(S, \{Q, D\}) = \{P, C\},$$

et

$$Z \notin Var(P) \cup Var(C).$$

Preuve. *Ce lemme est juste une extension de la propriété de définissabilité. Sa preuve est immédiate : il suffit de vérifier la propriété sur Z lors de l'induction.* ❑

Proposition A.1 *Soit Q une assertion telle que $Tmp_{i+1} \notin Var(Q)$. Si*

$$wp(S, \{Q, \bigwedge_{k=0}^{i+1} Tmp_{i+1}\}) = \{P, \bigwedge_{k=0}^{i+1} Tmp_{i+1}\},$$

alors

$$wp(\mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}, \{Q, \bigwedge_{k=0}^{i} Tmp_i\}) = \{P[B/Tmp_{i+1}], \bigwedge_{k=0}^{i} Tmp_i\}.$$

Preuve. *Soit $(\sigma, c) \in wp(\mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}, \{Q, \bigwedge_{k=0}^{i} Tmp_i\})$.*

Soit $(\sigma', c) = [\![\mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}]\!](\sigma, c)$. Nous avons $[\![\mathsf{S}]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c \wedge \sigma(B))$, et $(\sigma', c) \models \{Q, \bigwedge_{k=0}^{i} Tmp_i\}$ par la définition de wp. Soit $\sigma_1 = \sigma[Tmp_{i+1} \leftarrow \sigma(B)]$, et $\sigma'_1 = \sigma_1[Tmp_{i+1} \leftarrow \sigma(B)]$. Comme Tmp_{i+1} est une variable auxiliaire, nous avons $Tmp_{i+1} \notin Var(S)$, et

$$[\![\mathsf{S}]\!](\sigma_1, c \wedge \sigma(B)) = (\sigma'_1, c \wedge \sigma(B)),$$

et, comme $Tmp_{i+1} \notin Var(Q)$,

$$(\sigma'_1, c) \models \{Q, \bigwedge_{k=0}^{i} Tmp_i\}.$$

De plus, $\sigma'_1(Tmp_{i+1}) = \sigma(B)$, donc

$$(\sigma'_1, c \wedge \sigma(B)) \models \{Q, \bigwedge_{k=0}^{i+1} Tmp_{i+1}\}.$$

Nous pouvons déduire que $(\sigma_1, c \wedge \sigma(B)) \models \{P, \bigwedge_{k=0}^{i+1} Tmp_{i+1}\}$. *Donc*

$$\sigma \models P[B/\, Tmp_{i+1}].$$

Comme Tmp_i *est une variable auxiliaire, nous avons* $\forall i,\ Tmp_i \notin Var(S)$, *donc*

$$\sigma'(\bigwedge_{k=0}^{i} Tmp_i) = c \quad \Rightarrow \quad \sigma(\bigwedge_{k=0}^{i} Tmp_i) = c.$$

Réciproquement, soit $(\sigma, c) \in \left[\!\left[\{P[B/\, Tmp_{i+1}], \bigwedge_{k=0}^{i} Tmp_i\}\right]\!\right]$, *et* $\sigma_1 = \sigma[Tmp_{i+1} \leftarrow \sigma(B)]$. *Nous avons*

$$[\![\textsf{where } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = (\sigma', c),$$

avec $[\![\mathsf{S}]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c \wedge \sigma(B))$.

Avec $\sigma_1' = \sigma_1[Tmp_{i+1} \leftarrow \sigma(B)]$, *nous avons aussi*

$$[\![\textsf{where } B \textsf{ do } S \textsf{ end}]\!](\sigma_1, c) = (\sigma_1', c),$$

où $[\![\mathsf{S}]\!](\sigma_1, c \wedge \sigma(B)) = (\sigma_1', c \wedge \sigma(B))$.

Comme $(\sigma, c) \in \left[\!\left[\{P[B/\, Tmp_{i+1}], \bigwedge_{k=0}^{i} Tmp_i\}\right]\!\right]$, $\sigma_1 \models P$, *et comme* $Tmp_{i+1} \notin Var(B)$, *nous avons* $\sigma_1(\bigwedge_{k=0}^{i+1} Tmp_{i+1}) = c \wedge \sigma(B)$. *Par hypothèse, nous avons donc*

$$(\sigma_1', c \wedge \sigma(B)) \models \{Q, \bigwedge_{k=0}^{i+1} Tmp_{i+1}\}.$$

Comme $Tmp_{i+1} \notin Var(Q)$, *nous concluons que*

$$\sigma' \models Q$$

De plus, $\forall i,\ Tmp_i \notin Var(S)$, *donc*

$$\sigma'(\bigwedge_{k=0}^{i} Tmp_i) = \sigma(\bigwedge_{k=0}^{i} Tmp_i) = c.$$

Ceci conclut la preuve de la proposition A.1. ❑

Nous pouvons maintenant prouver la réciproque de la proposition d'équivalence.

Preuve. *Supposons que*

$$\vdash^* \{P, \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}.$$

Nous voulons trouver une preuve par annotations de la forme

$$\{P\}\ S^*\ \{Q\}.$$

Nous construisons celle-ci par induction sur la structure de S.

- *Si* $S \equiv X := E$ *: par la correction du système de preuve, nous avons*

$$\models \{P, \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}.$$

Par la définition de wp, nous avons

$$\{P, \bigwedge_{k=0}^{i} Tmp_i\} \Rightarrow wp(S, \{Q, \bigwedge_{k=0}^{i} Tmp_i\})$$

où $wp(S, \{Q, \bigwedge_{k=0}^{i} Tmp_i\}) = \{Q[\bigwedge_{k=0}^{i} Tmp_i ? E : X/X], \bigwedge_{k=0}^{i} Tmp_i\}$. *Alors*

$$\begin{array}{l} \{P\} \\ \{Q[\bigwedge_{k=0}^{i} Tmp_i ? E : X/X]\} \\ \quad S \\ \{Q\} \end{array}$$

est une preuve par annotations pour S.

- *Si* $S \equiv S_1 ; S_2$. *Soit*

$$\{P_2, \bigwedge_{k=0}^{i} Tmp_i\} = wp(S_2, \{Q, \bigwedge_{k=0}^{i} Tmp_i\})$$

et

$$\{P_1, \bigwedge_{k=0}^{i} Tmp_i\} = wp(S_1, \{P_2, \bigwedge_{k=0}^{i} Tmp_i\}).$$

Comme $\forall j > i, Tmp_j \notin Var(S) \cup Var(Q)$, *le lemme A.1 garantit que* $\forall j > i, Tmp_j \notin Var(P_2)$. *Les prémices de la règle de composition séquentielle sont donc satisfaits. Par la correction de* $\vdash^*$, *nous avons* $\models \{P, \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}$, *donc par la définition de wp,*

$$P \Rightarrow P_1.$$

Alors

$$\begin{array}{l} \{P\} \\ \{P_1\} \\ \quad S_1 \\ \{P_2\} \\ \quad S_2 \\ \{Q\} \end{array}$$

est une preuve par annotations pour S.

- *Considérons maintenant le cas où* $S \equiv$ where B do T end. *Le calcul des plus faibles préconditions nous permet de construire une preuve*

$$\vdash^* \{P', \bigwedge_{k=0}^{i+1} Tmp_{i+1}\}\ T\ \{Q, \bigwedge_{k=0}^{i+1} Tmp_{i+1}\},$$

où

$$\{P', \bigwedge_{k=0}^{i+1} Tmp_{i+1}\} = wp(T, \{Q, \bigwedge_{k=0}^{i+1} Tmp_{i+1}\}).$$

Par hypothèse d'induction,

$$\begin{array}{l}\{P'\}\\ \quad T^*\\ \{Q\}\end{array}$$

est une preuve par annotations pour T.
Or la proposition A.1 donne

$$wp(S, \{Q, \bigwedge_{k=0}^{i} Tmp_i\}) = \{P'[B/Tmp_{i+1}], \bigwedge_{k=0}^{i} Tmp_i\}.$$

Donc, par la correction du système de preuve, nous avons

$$\models \{P, \bigwedge_{k=0}^{i} Tmp_i\}\ S\ \{Q, \bigwedge_{k=0}^{i} Tmp_i\}.$$

Nous concluons que $P \Rightarrow P'[B/Tmp_{i+1}]$ et que

$$\begin{array}{l}\{P\}\\ \{P'[B/Tmp_{i+1}]\}\\ \quad \textsf{where } B \textsf{ do } [Tmp_{i+1} \equiv B]\\ \quad\quad \{P'\}\\ \quad\quad T^*\\ \quad\quad \{Q\}\\ \quad\quad \textsf{end}\\ \{Q\}\end{array}$$

est une preuve par annotations pour S.

Ceci conclut la preuve de la proposition d'équivalence. □

1.3 Second langage d'assertions

Nous donnons finalement les preuves des lemmes de définissabiité relatifs au second langage d'assertions. Nous avons les lemmes suivants.

Lemme 5 (Affectation) *Soit X une variable, E une expression vectorielle, $\langle W, A\rangle$ une assertion. Il existe une assertion $\langle V, A'\rangle$ telle que*

$$wlp(X{:=}E, \langle W, A\rangle) = [[\langle V, A'\rangle]]\ .$$

Preuve. *Dans le cas où $X = A$, prenons une nouvelle variable A' distincte de X, et définissons $\langle W', A'\rangle$ comme la A-conversion de $\langle W, A\rangle$. Sinon, prenons $W' = W$ et $A' = A$. Prenons maintenant $V = W'[(A'?E{:}\ X)/X]$. Comme les assertions $\langle W, A\rangle$ et $\langle W', A'\rangle$ sont équivalentes, $wlp(X{:=}E, \langle W, A\rangle) = wlp(X{:=}E, \langle W', A'\rangle)$.*

Soit maintenant s dans $wlp(X{:=}E, \langle W', A'\rangle)$. Si $s = \bot$, alors $s \in [\![\langle V, A'\rangle]\!]$. Sinon, $s = (\sigma, c)$ pour un certain environnement σ et un certain contexte c. Soit $(\sigma', c) = [\![S]\!](\sigma, c)$. Par définition de s nous avons $\sigma'[A' \leftarrow c] \models W'$. Or σ' est $\sigma[X \leftarrow c?\sigma(E){:}\,\sigma(X)]$. Nous avons donc $\sigma[A' \leftarrow c][X \leftarrow \sigma[A' \leftarrow c](A'?E{:}\,X)] \models W'$. Par le lemme de substitution, nous concluons que $\sigma[A' \leftarrow c] \models V$.

Réciproquement, soit s dans $[\![\langle V, A'\rangle]\!]$. Si $s = \bot$, alors $s \in wlp(X{:=}E, \langle W', A'\rangle)$. Sinon, $s = (\sigma, c)$ pour un certain environnement σ et un certain contexte c. Soit $(\sigma', c) = [\![S]\!](\sigma, c)$: σ' est $\sigma[X \leftarrow c?\sigma(E){:}\,\sigma(X)]$. Nous avons $\sigma'[A' \leftarrow c] = \sigma[X \leftarrow c?\sigma(E){:}\,\sigma(X)][A' \leftarrow c] = \sigma[A' \leftarrow c][X \leftarrow \sigma[A' \leftarrow c](A'?E{:}\,X)]$. Par la règle de substitution, nous avons donc

$$\sigma'[A' \leftarrow c] \models W'.$$

❑

Lemme 6 (Conditionnement) *Soit S un programme, B une expression vectorielle booléenne, $\langle W, A\rangle$ une assertion telle que $C \notin Var(B) \cup Change(S)$, et A' une variable telle que $A' \notin Var(W)$. S'il existe un prédicat W' tel que*

$$wlp(S, \langle W, A'\rangle) = [[\langle W', A'\rangle]],$$

alors

$$wlp(\textsf{where } B \textsf{ do } S, \langle W, A\rangle) = [[\langle W'[C \wedge B/A'], A\rangle]]\ .$$

Preuve. *Comme dans le lemme précédent, le cas des états indéfinis ($\bot$) est trivial.*

Prouvons tout d'abord que $wlp(\textsf{where } B \textsf{ do } S, \{W, A\}) \subseteq [\![\{W'[A \wedge B/A'], A\}]\!]$. Soit $(\sigma, c) \in wlp(\textsf{where } B \textsf{ do } S, \{W, A\})$. Nous avons $[\![\textsf{where } B \textsf{ do } S]\!](\sigma, c) = (\sigma', c)$, où $(\sigma', c \wedge \sigma(B)) = [\![S]\!](\sigma, c \wedge \sigma(B))$. Par définition, $(\sigma', c) \models \{W, A\}$. Dénotons $\sigma'[A \leftarrow c]$ par σ'_c, et $\sigma[A \leftarrow c]$ par σ_c. Nous avons $\sigma'_c \models W$. Comme $A' \notin Var(W)$, nous avons aussi $\sigma'_c[A' \leftarrow c \wedge \sigma(B)] \models W$, qui peut être réécrit en $(\sigma'_c, c \wedge \sigma(B)) \models \{W, A'\}$. Nous avons $A \notin Change(S)$, donc $[\![S]\!](\sigma_c, c \wedge \sigma(B)) = (\sigma'_c, c \wedge \sigma(B))$, et donc $(\sigma_c, c \wedge \sigma(B)) \models \{W', A'\}$. Nous réécrivons ceci en $\sigma_c[A' \leftarrow c \wedge \sigma(B)] \models W'$. Comme $A \notin Var(B)$, nous avons $\sigma_c[A' \leftarrow \sigma_c(A \wedge B)] \models W'$, et finalement $\sigma_c \models W'[A \wedge B/A']$ par le lemme de substitution.

Nous concluons que $(\sigma, c) \models \{W'[A \wedge B/A'], A\}$.

L'autre inclusion est prouvée exactement de la même façon.

❑

Lemme 7 (Itération) *Soit T un programme et $\langle W, A\rangle$ une assertion. Il existe une assertion $\langle V, A'\rangle$ telle que*

$$[[\langle V, A'\rangle]] = wlp(\textsf{loop } B \textsf{ do } T, [\![\langle W, A\rangle]\!]).$$

Preuve. *La preuve commence avec une description méta-mathématique de wlp. Un prédicat dans le langage d'assertions est construit par un codage à la Gödel. Une preuve complète serait très longue, nous ne donnons donc ici que le point de départ et les arguments principaux.*

Soit S un programme $\mathcal{L}$, $S \equiv \textsf{loop } B \textsf{ do } T$. Nous commençons par montrer qu'un état (σ, c) est dans la plus faible précondition de la boucle $wlp(\textsf{loop } B \textsf{ do } T, \{W, A\})$ si et seulement si $\sigma^c = \sigma[A' \leftarrow c]$ $(A' \notin Var(S))$ satisfait la propriété suivante :

$$\forall k : \forall \sigma_0, \sigma_1, \ldots, \sigma_k : \begin{cases} Si & \sigma^c = \sigma_0 \\ et & \forall i \in [0, k[: \begin{cases} \sigma_i \models \exists u : B|_u \wedge A'|_u \\ [\![T]\!]((\sigma_i, \sigma_i(A'))) = (\sigma_{i+1}, \sigma_{i+1}(A')) \end{cases} \\ alors & \sigma_k \models (W \wedge A = A') \vee (\exists u : B|_u \wedge A'|_u) \end{cases}$$

Dénotons par $\mathcal{V}$ cette propriété. Celle-ci correspond à la description dénotationnelle de l'itération. Comme chaque environnement σ_i peut être dénoté par un entier s_i (il y a un nombre fini de vecteurs finis), la seconde étape montre que $[\![T]\!]((\sigma_i, \sigma_i(A'))) = (\sigma_{i+1}, \sigma_{i+1}(A'))$ est équivalent à

$$(\sigma_i, c) \in (wlp(T, \{\bar{X} = s_{i+1}, A\}) \wedge \neg wlp(T, \{false, A\}))$$

où $\bar{X}$ dénote toutes les variables qui sont définies dans l'environnement et $[\![\{\bar{X} = s_{i+1}, A\}]\!] = \{\sigma_{i+1}\}$. Par hypothèse d'induction, cette plus faible précondition peut être exprimée par une assertion. Il est donc possible de définir un prédicat V' tel que $\mathcal{V}$ est équivalent à $\forall k : \forall s_0, s_1, \ldots, s_k : V'$.

La dernière étape de la preuve consiste à coder la suite d'entiers $s_0, s_1, \ldots, s_k$ par un prédicat de Gödel. Cette propriété $\mathcal{W}$ est alors dénotée par $\forall k : \forall s : V''$, qui est notre prédicat final V.

Le lecteur intéressé pourra trouver davantage de détails sur ce type de preuve dans [25], ou une introduction dans [70].

□

annexe **B**

Compléments sur ALPHA

Nous donnons ici la preuve de la proposition 19 du chapitre 6. Rappelons tout d'abord les notations. Nous avons un système S appelant un module S'. On note

- $\overline{S}$ le programme S dans lequel on a substitué l'appel au module par le module lui-même, après renommage et extension ; $\overline{S}$ est défini à renommage près des variables locales au module ;
- $In(S), Out(S), Loc(S), Var(S)$ les ensembles respectifs de variables d'entrée, de sortie, locales de S, et l'union de ces ensembles ;
- de même pour $In(S'), \ldots$ et $In(\overline{S}), \ldots$, que nous noterons également $In', \ldots$ et $\overline{In}, \ldots$; on a $In(\overline{S}) = In(S), Out(\overline{S}) = Out(S)$ et $Var(\overline{S}) = Var(S) \cup Loc(S')$.

Nous considérons que V est un sous-ensemble de Out, et nous notons $W = \mathit{Nécessaire}(V)$. Nous noterons Σ un état de S, σ un état de S' et Γ un état de $\overline{S}$.

Les transitions du système S par rapport à V sont notées $\underset{V}{\Longrightarrow}_S$.

On pose

$$\overline{V} = V$$

et

$$\overline{W} = W \cup \mathit{Nécessaire}_{S'}(V')$$

Nous voulons montrer la proposition suivante

Proposition 19 (Équivalence des grands pas avec la substitution) *Si Σ et Γ sont des états initiaux respectivement de S et de $\overline{S}$ équivalents sur $W \cap In$, et si*

$$\Sigma \underset{V}{\overset{*}{\Longrightarrow}}_S \Sigma' \text{ avec } V \subseteq supp(\Sigma')$$

alors

$$\Gamma \underset{V}{\overset{*}{\Longrightarrow}}_{\overline{S}} \Gamma' \quad avec \quad V \subseteq supp(\Gamma')$$
$$et \quad \Gamma' \underset{W}{\sim} \Sigma'$$

et réciproquement.

La preuve de cette proposition fait appel à deux lemmes préliminaires concernant l'ensemble $\overline{W}$.

Lemme B.1 *On a*

- $\overline{W} \cap Var(S) \subseteq W$

- $\overline{W}$ *est clos à gauche pour* $\overline{S}$

Lemme B.2 *Les variables de* $\overline{W} \setminus (Loc(S') \cup Out')$ *ne dépendent directement que de variables de* $\overline{W} \setminus Loc(S')$.

Les preuves de ces lemmes découlent immédiatement des définitions.
Nous donnons maintenant l'essentiel de la preuve de la proposition.

Preuve. *Nous commençons par le sens direct. L'idée est de construire à partir de la suite des états* Σ *une autre suite d'états* T *qui constitue une suite de transitions de* $\overline{S}$. *Considérons donc les états* Σ_i *successifs, avec* $\Sigma_0 = \Sigma$ *et* $\Sigma_n = \Sigma'$. *Posons* $\Gamma_0 = \Sigma$, *et supposons que nous avons déjà examiné les transitions* $\Sigma_0 \Rightarrow_S \Sigma_1 \Rightarrow_S \ldots \Rightarrow_S \Sigma_{i-1} \Rightarrow_S \Sigma_i$ *et construit la suite* $\Gamma_0, \Gamma_1, \ldots \Gamma_j$ *à partir de ces états. Examinons maintenant la transition* $\Sigma_i \Rightarrow \Sigma_{i+1}$. *Deux cas se présentent :*

- *c'est une transition « classique » qui correspond à l'évaluation d'une variable* x *de* $Var(S) \setminus Var(S')$ *; on pose alors*

$$\Gamma_{j+1} = \Gamma_j[x \leftarrow \Sigma_{i+1}(x)]$$

- *c'est une transition définie par une séquence* $\sigma_0 \underset{V'}{\overset{*}{\Longrightarrow}}_{S'} \sigma_m$ *de* S' *; on pose alors* $j_0 = j$, *et pour tout* $k \in \{1, .., m\}$, *si* $\sigma_{k-1} \underset{V'}{\Longrightarrow}_{S'} \sigma_k$ *correspond à l'évaluation d'une variable* x_k *telle que* $\Gamma_{j_{k-1}}(x_k) = \perp$, *on pose*

$$j_k = j_{k-1} + 1 \text{ et } \Gamma_{j_k} = \Gamma_{j_{k-1}}[x_k \leftarrow \sigma_k(x_k)]$$

sinon (si x_k *est déjà définie dans* $\Gamma_{j_{k-1}}$*) on pose*

$$j_k = j_{k-1}$$

Intuitivement, Γ_j *est successivement enrichi par les calculs des transitions internes au module correspondant à des variables non encore définies dans* Γ_j.

On a alors construit $\Gamma_j, \ldots, \Gamma_{j+l}$ *; on examine la transition* $\Sigma_{i+1} \Rightarrow \Sigma_{i+2}$ *de la même façon. Notons* $\varphi(i)$ *l'indice* j *de* Γ_j *lorsque l'on a examiné les états jusqu'à* Σ_i. φ *est une fonction strictement croissante de* i.

Il nous faut maintenant prouver que les Γ_j *peuvent définir une suite de transitions de* $\overline{S}$. *Remarquons tout d'abord le fait suivant, qui exprime que les états* Σ *et* T *coïncident sur les variables de* S *:*

$$\Sigma_i \underset{W}{\sim} \Gamma_{\varphi(i)}$$

Ceci se prouve par récurrence sur i. *Le cas* $i = 0$ *est trivial. Traitons le cas où ce fait est déjà établi pour* i.

- *Si la transition est « petits-pas », le résultat est immédiat.*
- *Si la transition est induite par une suite* $\sigma_0 \underset{V'}{\overset{*}{\Longrightarrow}}_{S'} \sigma_m$, *on a*

$$\Sigma_i \underset{W}{\sim} \Gamma_{\varphi(i)}$$
$$\Sigma_{i+1} = \Sigma_i[x \leftarrow \sigma_m(x)]$$
$$\Gamma_{\varphi(i+1)}(x) = \sigma_m(x)$$

Les transitions $\Gamma_{\varphi(i)} \to \Gamma_{\varphi(i)+1} \to \ldots \to \Gamma_{\varphi(i+1)}$ *valuent des variables de* S'. *Comme il n'y a pas de dépendance entre variables de* Out', *les variables de* W *concernées sont donc des variables de* In'. *Celles-ci sont déjà toutes valuées dans* Σ_i. *Le lemme de conservation s'applique, qui affirme que leurs valeurs ne sont pas modifiées.*

On a donc bien

$$\Sigma_{i+1} \underset{W}{\sim} \Gamma_{\varphi(i+1)},$$

ce qui conclut cette preuve par récurrence.

Nous pouvons maintenant attaquer le corps de la preuve : montrer que les Γ_j *forment une suite de transitions de* $\overline{S}$. *Par construction, les conditions de définition des variables sont bien vérifiées. Considérons une transition* $\Gamma_j \overset{x_k}{\Longrightarrow} \Gamma_{j+1}$ *et montrons que* $\Gamma_{j+1}(x_k) = \Gamma_j(e_x(\vec{y}))$, *où* $x = e_x(\vec{y})$ *est l'expression de définition de* x_k.

- *si c'est une transition « classique », le résultat est immédiat grâce au lemme B.2 ;*
- *si c'est une transition induite par* $\sigma_0 \underset{V'}{\overset{*}{\Longrightarrow}}_{S'} \sigma_m$, *on a* $\Gamma_{j+1}(x_k) = \sigma_{k+1}(x_k) = \sigma_k(e_x(\vec{y}))$.
  Montrons que $\Gamma_j(y) = \sigma_k(y)$ *pour tout* y *de* $\vec{y}$. *Il nous donc faut retrouver le « moment » où* y *a été valuée.*
  Soit $i_0 = \max\{i \mid \varphi(i) \leq j\}$. *On a*

  $$\Gamma_{\varphi(i_0)} \underset{W}{\sim} \Sigma_{i_0}$$

  donc en particulier

  $$\Gamma_{\varphi(i_0)} \underset{W \cap In'}{\sim} \sigma_0.$$

  Deux cas se présentent alors
  - *si* y *a été valuée au cours de la suite* $\Gamma_{\varphi(i_0)} \ldots \Gamma_j$, *par exemple entre* Γ_p *et* Γ_{p+1} *avec* $\varphi(i_0) \leq p < j$. *Ceci correspond à une transition* $\sigma_q \Rightarrow \sigma_{q+1}$ *avec* $0 \leq q < k$. *On a alors*

    $$\Gamma_j(y) = \Gamma_{p+1}(y) = \sigma_{q+1}(y) = \sigma_k(y)$$

    par conservation de la valeur entre $p+1$ *et* j *d'une part,* $q+1$ *et* k *d'autre part.*
  - *si* y *a été valuée avant* $\Gamma_{\varphi(i_0)}$; *à nouveau, deux cas se présentent*
    - *soit* $y \in In'$; *on a alors*

      $$\Gamma_j(y) = \Gamma_{\varphi(i_0)}(y) \text{ par conservation de la valeur}$$
      $$\Gamma_{\varphi(i_0)}(y) = \sigma_0 \text{ par équivalence sur } In' \cap W$$
      $$\sigma_k(y) = \sigma_0(y) \text{ par conservation de la valeur}$$

      donc on a bien $\Gamma_j(y) = \sigma_k(y)$;
    - *soit* $y \notin In'$. *Il nous faut retrouver d'où provient sa valeur.* y *a été valué dans une suite* $\Gamma_{\varphi(i'_0)} \ldots \Gamma_{p'+1}$ *avec* $i'_0 = \max\{i \mid \varphi(i) \leq p+1\}$. *On a évidemment* $i'_0 < i_0$. *Cette suite est définie par* $\sigma'_0 \ldots \sigma'_{m'}$ *et* $\Gamma'_{p+1}(y) = \sigma'_{q'+1}(y)$. *On a*

      $$\sigma'_0 \underset{W \cap In'}{\sim} \Gamma_{\varphi(i'_0)}$$
      $$\sigma_0 \underset{W \cap In'}{\sim} \Gamma_{\varphi(i_0)}$$

donc par conservation de la valeur

$$\sigma'_0 \underset{W \cap In'}{\sim} \sigma_0$$

qui signifie que les états initiaux sont les mêmes à chaque appel du module. On a enfin

$$\sigma'_{q'+1}(y) = \sigma_k(y) \text{ par déterminisme}$$
$$\Gamma'_{p+1}(y) = \Gamma_j(y) \text{ par conservation de la valeur}$$
$$\Gamma_j(y) = \sigma_k(y) \text{ par transitivité}$$

Ceci termine la preuve du sens direct de la proposition. Nous esquissons maintenant la preuve de la réciproque. La preuve se fait de façon similaire, mais avec une construction inverse. On examine les transitions $\Gamma_i \Rightarrow \Gamma_{i+1}$ *et on construit une suite* Σ_j *:*

- *si* $\Gamma_i \Rightarrow \Gamma_{i+1}$ *évalue une variable* x *de* $\overline{W} \backslash (Loc' \cup Out')$*, on construit* Σ_{j+1} *à partir de* Σ_j *en l'enrichissant avec la valeur de* x *;*
- *si* $\Gamma_i \Rightarrow \Gamma_{i+1}$ *évalue une variable* x *de* Loc'*, on n'en tient pas compte ;*
- *si* $\Gamma_i \Rightarrow \Gamma_{i+1}$ *évalue une variable* x *de* Out'*, on montre que l'on est capable de construire un « grand pas ». En effet, les variables de* In' *et de* Loc' *nécessaires au calcul de* x *sont valuées. La preuve que la valeur ainsi obtenue pour* x *est la même que dans* Γ_{i+1} *est semblable à celle de l'équivalence des deux sémantiques opérationnelles au chapitre 3.*

Ceci termine la preuve de la proposition. ❑

Résumé : Dans cette thèse, nous nous sommes intéressé à la validation formelle des langages à parallélisme de données. L'idée est de tirer parti de la relative simplicité de ce modèle de programmation pour développer des méthodes semblables à celles déjà éprouvées dans le cadre des langages scalaires classiques. La première partie du travail effectué concerne un langage data-parallèle simple, de type impératif. Nous avons montré qu'il était possible de définir un système de preuve complet pour ce langage, inpiré de la logique de Hoare. L'étude théorique nous a permis en outre de définir une méthodologie pratique de preuve par annotations, semblable à celle utilisée pour les langages scalaires. Nous nous sommes ensuite tourné vers le langage d'équations récurrentes Alpha. Il s'avérait nécessaire de définir pour ce langage un cadre formel de validation, plus riche que le système de transformations existant ne permettant que des preuves par équivalence. Nous avons défini un modèle d'exécution par l'intermédiaire d'une sémantique opérationnelle, et une méthodologie de preuve. Celle-ci utilise des invariants qui sont raffinés à partir d'une traduction du programme dans un langage logique jusqu'à l'obtention de la propriété voulue.

Mots-clés : Programmation parallèle, spécification et validation de programmes, sémantique des langages de programmation, langages data-parallèles, équations récurrentes, méthode de preuve, logique de Hoare, plus faibles préconditions, invariants.

Abstract: In this thesis, we address the problem of formal validation for data parallel languages. The idea is to exploit the relative simplicity of this programming model to develop some methods, resembling those already in use for classical scalar languages. The first part of this work deals with a simple data parallel imperative language. We have shown that it was possible to define for this language a complete proof system, inspired from Hoare logic. This theoretical study has also be the key to define a practical methodology of proof by annotations, similar to the methodology used for scalar languages. We then worked on the Alpha language, that is a language of recurrence equations. It appeared that it was necessary to define a formal validation framework for this language, since the existing rewriting system allows only proofs of equivalence between programs. We have defined an execution model by means of an operational semantics, and a proof methodology. This methodology is based on invariants, that are refined from a translation of the program into a logical language to the final desired property.

Keywords: Parallel Programming, Specifying and Verifying and Reasoning about Programs, Semantics of Programming Langages, Data-Parallel Languages, Recurrence Equations, Proof Methods, Hoare Logic, Weakest Preconditions, Invariants.